pyfora Documentation

Release 0.5.8

Ufora Inc.

September 09, 2016

CHAPTER 1

Compiled, automatically parallel Python for data-science

Any code you run with pyfora works as-is in python, but with pyfora it can run hundreds or thousands of times faster, and can operate on datasets many times larger than the RAM of a single machine. You can speed up your computations by running them on hundreds of CPU cores with terabytes of RAM, and you can do this with hardly any changes to your code.

pyfora consists of two main components:

- A distributed backend that runs on one or more machines in your local network or in the cloud.
- A Python package that sends your code to the backend for compilation and execution.

1.1 Example

The following program uses pyfora to sum `math.sin()` over the first billion integers:

```
import math, pyfora
executor = pyfora.connect('http://localhost:30000')

with executor.remotely.downloadAll():
   x = sum(math.sin(i) for i in xrange(10**9))

print x
```

This program runs in **13.76 seconds** on a 3.40GHz Intel(R) Core(TM) i7-2600 quad-core (8 hyperthreaded) CPU, and utilizes all 8 cores. The same program in the local python interpreter takes **185.95 seconds** and uses one core.

1.2 Installation

```
pip install pyfora
```

pyfora requires Python 2.7. Python 3 is not supported yet.

Note: Only official CPython distributions from python.org are supported at this time. This is what OS X and most Linux distributions include by default as their "system" Python.

1.3 Backend Installation

The pyfora backend is distributed as a docker image that can be run in any docker-supported environment. The *Setup Guides* below contain instructions for setting up the backend in various environments.

1.3.1 Running pyfora on AWS

If you have an Amazon Web Services account you can get pyfora running at scale within minutes. The pyfora package includes an auxiliary script called pyfora_aws, which helps get you started on AWS.

What You Need to Get Started

AWS Account

You'll need an AWS account with an access key that has permission to launch EC2 instances. If you don't yet have an access key, follow these instructions to create one.

boto

The pyfora_aws tool uses boto to communicate with AWS:

```
pip install boto
```

Launch the Backend

Credentials

pyfora_aws uses boto to interact with EC2 on your behalf. If you already have a Boto configuration file with your credentials then no additional configuration in needed. Otherwise, you can set your credentials using the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.

To set the envrionment variables, open a terminal window and type:

```
export AWS_ACCESS_KEY_ID=<your aws access key id>
export AWS_SECRET_ACCESS_KEY=<your aws secret key>
```

SSH Key-Pair

While not strictly required, it is strongly recommended that you register an SSH key-pair with EC2 and use it when launching instances. Otherwise, you will not be able to log in to the launched instances for diagnostics and troubleshooting. See Amazon EC2 Key Pairs for more information.

Note: This tutorial assumes that you **are** providing an SSH key and uses SSH to tunnel traffic to/from launched instances. If you do not wish to use an SSH key, or tunnel HTTP traffic over SSH, please the the reference documentation for pyfora_aws.

Start a Backend Instance

You are now ready to start some instances using pyfora_aws. The following command will launch and configure a single c3.8xlarge on-demand instance in the us-east-1 region. It takes about 5-6 minutes to complete:

```
$ pyfora_aws start --ssh-keyname <name_of_your_SSH_keypair>

Launching manager instance:
Mon Mar  7 10:24:42 2016 -- i-4aef5b89: pending /
Done

Manager instance started:

    i-4aef5b89 | 184.169.200.155 | running | manager

To tunnel the pyfora HTTP port (30000) over ssh, run the following command:
    ssh -i <ssh_key_file> -L 30000:localhost:30000 ubuntu@184.169.200.155

Waiting for services:

Mon Mar  7 10:26:20 2016 -- Instance:i-4aef5b89: installing dependencies -
Mon Mar  7 10:29:10 2016 -- Instance:i-4aef5b89: installing docker 1.9 -
Mon Mar  7 10:30:28 2016 -- Instance:i-4aef5b89: pulling docker image -
Mon Mar  7 10:30:51 2016 -- Instance:i-4aef5b89: launching service -
Mon Mar  7 10:30:52 2016 -- Instance:i-4aef5b89: ready
Done
```

Where `<name_of_your_SSH_keypair>` is the name you gave your SSH key-pair in EC2.

SSH Tunnelling

By default, to keep things secure, pyfora_aws keeps all ports on launched instances inaccessible to incoming connections, with the exception of port 22 for SSH connections. The easiest secure way to connect to the launched instance from your machine is by tunnelling pyfora's HTTP port - 30000 - over SSH. This means that all traffic between your machine and the instance is secured by SSH.

To establish a tunnel, open a new terminal window (it will need to stay open for the duration of your session) and run:

```
$ ssh -i <ssh_key_file> -L 30000:localhost:30000 ubuntu@<manager_ip_address>
```

Where `<ssh_key_file>` is the path to the private key file of the SSH key-pair you specified when launching the instance, and `<manager_ip_address>` is the public IP address of the manager machine (184.169.200.155 in the example above).

The `-L` option tells SSH to map port `30000` on your local machine to `localhost:30000` on the remote.

Connect to the Backend

Now that the SSH tunnel is open you can connect to the backend using `localhost:30000`. To verify your connection, copy the code below to a new `test_pyfora.py` file:

```
import pyfora
executor = pyfora.connect('http://localhost:30000')

with executor.remotely.downloadAll():
    x = sum(xrange(10**9))
```

```
print x
```

And run it in your terminal:

```
$ python test_pyfora.py
499999999500000000
```

Adding Instances

If you need more compute power you can easily increase the size of your cluster by launching additional instances. The following command add two more c3.8xlarge instances to your running backend:

```
$ pyfora_aws add -n 2
Tue Mar  7 10:52:57 2016 -- pending (2) /
Tue Mar  7 10:53:04 2016 -- running (1), pending (1) \
Done

Workers started:
    i-3c9324ff | 54.219.34.156 | running | worker
    i-149225d7 | 54.219.31.180 | running | worker

Waiting for services:

Tue Mar  7 10:54:20 2016 -- installing dependencies (2) -
Tue Mar  7 10:54:37 2016 -- installing dependencies (1), installing docker 1.9 (1) \
Tue Mar  7 10:57:09 2016 -- installing docker 1.9 (2) \
Tue Mar  7 10:58:04 2016 -- installing docker 1.9 (1), pulling docker image (1) /
Tue Mar  7 10:58:37 2016 -- pulling docker image (2) -
Tue Mar  7 10:58:41 2016 -- launching service (1), pulling docker image (1) /
Tue Mar  7 11:00:01 2016 -- ready (1), pulling docker image (1) -
Tue Mar  7 11:00:17 2016 -- ready (1), launching service (1) |
Tue Mar  7 11:00:18 2016 -- ready (2)
Done
```

Stopping Instances

To terminate all instances in your cluster run:

```
$ pyfora_aws stop --terminate
Terminating 3 instances:
    i-3c9324ff | 54.219.34.156 | running | worker
    i-799423ba | 54.176.73.201 | running | manager
    i-149225d7 | 54.219.31.180 | running | worker
```

1.3.2 Running pyfora on a Single Box

You can easily run the pyfora backend locally on your machine using docker. Then you can connect pyfora to your local backend and start using it to speed up your python code.

What You Need to Get Started

OS

You'll need an OS that can run docker. Currently this means:

- A reasonably recent version of a 64-bit Linux distribution such as: Ubuntu, Debian, RedHat, Fedora, Centos, Gentoo, Suse, Amazon, Oracle, etc.
- OS X 10.8 "Mountain Lion" or newer.
- Windows 7.1, 8/8.1 or newer.

Note: Docker only runs *natively* on Linux. On Windows and OS X it runs in a virtual machine. As a result, services running in docker containers need to be addressed using the VM's IP address instead of `localhost`. The code examples in this tutorial use `localhost` and should be adjusted when running in non-Linux environments.

Docker

Docker is available for Linux, OS X, or Windows. To install docker on your machine, visit http://www.docker.com/, click the Get Started link, and follow the instructions.

On Linux you can also, optionally, follow these instructions to enable running docker commands without sudo. Note, however, that the docker daemon still runs as root - it just saves you five keystrokes when running docker commands.

Pull the pyfora Service Image

Once docker is installed you can pull the latest backend service image.

```
$ sudo docker pull ufora/service:latest
```

Important: If you are not using the most recent pyfora release and don't want to upgrade, you will need to use a docker image compatible with your version. For example, if you are using pyfora version 0.3.1, you can pull and use the `ufora/service:0.3.1` image.

Start the Backend Container

The command below starts an all-in-one docker container that runs all the backend services needed to support pyfora. To run a cluster on multiple machines in a local network, follow the instructions here.

Create a local directory for the backend services logs:

```
$ mkdir ~/ufora
```

And run (replacing the path `/home/user` with your own home directory path):

```
$ sudo docker run -d --name pyfora -p 30000:30000 -v /home/user/ufora:/var/ufora ufora/service
```

What does this do?

- `docker run` launches a new docker container.
- `-d` starts the container as daemon that runs in the background.
- `--name pyfora` names the new container pyfora for easy reference in subsequent commands.
- `-p 30000:30000` maps port 30000 - pyfora's default HTTP port - to the same port number in your host OS. This lets pyfora connect to the container using `http://localhost:30000`.
- `-v /home/user/ufora:/var/ufora` mounts the local directory ~/ufora into /var/ufora within the container. This is where Ufora writes all of its log files.
- `ufora/service` is the name of the Ufora service image to run. To use a version other than the latest, specify a version tag (e.g. `ufora/service:0.2`).

Verify

With your backend container running, you can now verify that pyfora is able to connect to it. Open `python` in your terminal and run:

```
>>> import pyfora
>>> pyfora.connect('http://localhost:30000')
<pyfora.Executor.Executor object at 0x7f518a7c1c90>
```

If no exceptions are thrown, you have a working pyfora cluster running on your machine!

This would be a good point to jump over to the Introduction to pyfora tutorial and learn more about coding with pyfora.

Stopping the pyfora Container

When you want to stop the pyfora service container, run:

```
$ sudo docker stop pyfora
```

This stops the container but preserves its state so it can be restarted at a later time. To permanently delete the container and all its state, run the following command after stopping the container:

```
$ sudo docker rm pyfora
```

To restart the container after it has been stopped:

```
$ sudo docker start pyfora
```

1.3.3 Setting up a Cluster on a Local Network

Before You Begin

This tutorial walks you through installing the pyfora backend on a cluster of machines. If you have not read through the Running pyfora on a Single Box tutorial yet, it is recommended that you familiarize yourself with it before continuing with the multi-machine setup.

pyfora Cluster Topology

A pyfora cluster consists of a single *manager* and one or more *workers*. Workers contribute CPU and memory resources to the cluster and are where all computations take place. Workers connect to the manager and register themselves with it. They use the manager to discover each other's network addresses and port configuration and to find out when new workers join the cluster.

The manager, in addition to acting as a registry of workers, also acts as the cluster's front end. Client machines that use the pyfora package to submit computations to a cluster only ever talks to the cluster's manager. Workers only communicate with each other and with their manager. The pyfora package connects to the manager over HTTP using socket.io to support real-time notifications from the cluster.

The Backend Docker Image

In the Running pyfora on a Single Box tutorial you used the `ufora/service` docker image to start a container that ran both the manager and a worker on your local machine. The same image can be configured to run a worker that connects to a specified manager or, optionally, run the manager without a worker.

Environment Variables

There are several environment variable that can be set when launching a pyfora container to configure its behavior.

- `UFORA_MANAGER_ADDRESS`: the host-name or IP address of the manager. Setting this variable causes the container to only run the worker service. Without this variable, the container runs both manager and worker services.
- `UFORA_WORKER_OWN_ADDRESS`: the host-name or IP address that the worker uses to register itself with the manager. The manager and other workers try to connect to it using this address. This is useful in situations where you have multiple network interfaces (public and private, or a docker container running in bridge mode) and you want to tell the worker which address to register. The variable is required unless the worker container is started with the `--net=host` option, in which case the worker tries to figure out its own address using `socket.gethostbyname(socket.getfqdn())` if the variable isn't set.
- `UFORA_WORKER_BASE_PORT`: the first of two consecutive ports that the worker listens on. This is useful if you want to run multiple workers side-by-side.
- `UFORA_NO_WORKER`: Set this variable to 1 to prevent the manager container from also running a worker. This variable and `UFORA_MANAGER_ADDRESS` are mutually exclusive. At most one of them can be set.
- `UFORA_WEB_HTTP_PORT`: the port used by the manager's HTTP server.

Ports

Worker Ports pyfora workers communicate with each other over two consecutively numbered ports. One port is used to maintain a control channel over which they coordinate work, and the other is used as a data channel where large chunks of data can be transmitted.

The default ports are: `30009` and `30010`.

They can be configured using the `UFORA_WORKER_BASE_PORT` environment variable.

Manager Ports The manager listens on two ports. One is the worker registry service to which workers connect, and the other is the HTTP server that clients connect to using the pyfora package.

The worker registry port is 30002 and is not currently configurable. A configuration option will be added in a future release. This port only needs to be accessible to workers.

The default HTTP port is `30000` and is configured using the `UFORA_WEB_HTTP_PORT` environment variable.

Security If you run the cluster on a local, trusted network you may not need to worry about this and can skip to the next section. If, however, you run your cluster in the cloud or a shared network, you may want to read on.

The pyfora services do not have any build-in authentication mechanisms. There is no notion of accounts, credentials, logging-in, etc. If you have network access to the services, you can submit work. It is therefore recommended that you configure firewall rules (or a security group on AWS) such that only machines in the cluster can connect to your workers on their ports (`30009`, and `30010` by default), and to your manager on the worker-registry port (`30002`).

To connect your pyfora client in a secure way, it is recommended that you tunnel your HTTP traffic over SSH using the `-L port:host:hostport` option. For example, if your manager is running at `54.144.209.248` you can map your local port `30000` to the same port on the manager using:

```
$ ssh user_name@54.144.209.248 -L 30000:localhost:30000
```

Now as long as your SSH session is open, you can connect to the manager using `localhost:30000`.

Running the Service

The instructions below assume you have already installed docker and pulled the `ufora/service` image on all machines in the cluster.

While not strictly necessary, it is recommended that you create a directory on all your machines which will be mounted to `/var/ufora` on all your pyfora containers. The pyfora services will write their logs into it, and having it on the host machine can make accessing logs easier. The instructions below assume this directory is `/home/user/ufora`, replace it with your own path when running the commands.

The Manager

Pick a machine to run the manager service and run the following command to start the manager and a worker on it:

```
$ sudo docker run -d --name pyfora_manager -p 30000:30000 -p 30002:30002 -v /home/user/ufora:/var/ufo
```

To run the manager service without a worker run:

```
$ sudo docker run -d --name pyfora_manager -e UFORA_NO_WORKER=1 -p 30000:30000 -p 30002:30002 -v /hom
```

Workers

If your manager is running, for example, at `192.168.1.15`, and the worker is at `192.168.2.11`, start a worker using:

```
$ sudo docker run -d --name pyfora_worker -e UFORA_MANAGER_ADDRESS=192.168.1.15 -e UFORA_WORKER_OWN_A
```

Repeat this on every machine you want to use as a worker in your cluster.

Verify

You can now verify that pyfora is able to connect to the manager and run computations. Create a local file called `testpyfora.py` with the following conent:

```
import pyfora, math

executor = pyfora.connect('http://<your_manager_address>:30000')
with executor.remotely.donwloadAll():
    x = sum(math.sin(i) for i in xrange(10**9))

print x
```

Run it from your terminal:

```
$ python testpyfora.py
-0.124548962703
```

This may be a good point to jump over to the Introduction to pyfora tutorial and learn more about coding with pyfora.

1.3.4 Introduction to pyfora

pyfora sends your Python code to a local or remote cluster, where it is compiled and can be automatically run in parallel on many machines. With pyfora you can run distributed computations on a cluster without having to modify your code. You can speed up your computations by running them on hundreds of CPU cores, and you can operate on datasets many times larger than the RAM of a single machine.

Setting up a Cluster

To get started with pyfora you will need a backend cluster with at least one worker. The backend is available as a docker image that can be run locally on your machine in a single-node setup, or on a cluster of machines on a local network or in the cloud.

Connecting to a Cluster

Once you have a running cluster, you can connect to it and start executing code:

```
import pyfora
executor = pyfora.connect('http://localhost:30000')
```

The variable `executor` is now bound to a pyfora *`Executor`* that can be used to submit and execute code on the cluster. There are two ways to run code with an *`Executor`*:

1. **Asynchronously** using *`Future`* objects.
2. **Synchronously** by enclosing code to be run remotely in a special python `with` block.

In this tutorial we will use the synchronous model.

First, we'll define a function that we are going to work with:

```
def isPrime(p):
    x = 2
    while x*x <= p:
        if p%x == 0:
            return 0
        x = x + 1
    return 1
```

Computing Remotely

Now we can use the *Executor* to run some remote computations with the function we defined:

```
with executor.remotely.downloadAll():
    result = sum(isPrime(x) for x in xrange(10 * 1000 * 1000))

print result
```

What just happened?

The code in the body of the `with` block was shipped to the cluster, along with any dependent objects and code (like `isPrime()` in this case) that it refers to, directly or indirectly.

The python code was translated into pyfora bitcode and executed on the cluster. The resulting objects (`result` in the example above) were downloaded from the cluster and copied back back into the local python environment because we used *executor.remotely.downloadAll()*.

Depending on the code you are running, and the number of CPU cores available in your cluster, the runtime looks for opportunities to parallelize the execution of your code. In the example above, the runtime can see that individual invocations of `isPrime()` within the generator expression `isPrime(x) for x in xrange(10 * 1000 * 1000)` are independent of each other can therefore be run in parallel.

In fact, what the runtime does in this example is to split the `xrange()` iteration across all available cores in the cluster. If a particular subrange completes while others are still running, the runtime dynamically subdivides a range that is still computing to maximize the utilization of CPUs. This is something that is bound to happen in problems like this when time-complexity of a computation is not constant across the entire input space (determining whether a large number is prime is much harder than a small number).

Important: Not **all** python code can be converted to pyfora bitcode and run in this way. In order to benefit form the performance and scalability advantages of pyfora, your code must be:

1. **Side-effectless**: data structures cannot be mutated.
2. **Deterministic**: running with the same input must always yield the same result.

See *Pure Python* for more details.

Working with proxies

In the previous example, the result of the computation was the number of prime numbers in the specified range. That's a single `int` that can be easily downloaded from the cluster and copied into the local python environment.

Now consider this variation of the code:

```
with executor.remotely.remoteAll():
    primes = [x for x in xrange(10 * 1000 * 1000) if isPrime(x)]
```

This time the result of the computation, the variable `primes`, is a `list` of all prime numbers in the range. But because we used *executor.remotely.remoteAll()*, the variable `primes` is a *proxy* to a list of primes that lives in-memory on the cluster (it is actually an instance of *RemotePythonObject*).

There are two things you can do with remote python objects:

1. Download them into the local python scope where they become regular python objects.
2. Use them in subsequent remote computations on the cluster.

Downloading a remote object is done using its *toLocal()* method, which returns a *Future* that resolves to the downloaded object. To do it all synchronously in one line you can write:

```
primes = primes.toLocal().result()
```

This call downloads all the data in the remote `primes` list from the cluster to the client machine where it is converted back into python. If the list is very large, or the connection to the cluster is slow, this can be a slow operation. Furthermore, the size of the list may be greater than the amount of memory available on the local machine, in which case it is impossible to download it this way.

As an alternative to downloading the entire result, we may choose to compute with it inside of another `with executor.remotely` block. For example:

```
with executor.remotely.downloadAll():
    lastFewPrimes = primes[-100:]
```

The backend recognizes that `primes` is a proxy to data that lives remotely in the cluster, and lets us refer to it in dependent computations, the result of which we then return as regular python objects.

For convenience, it also possible to write:

```
with executor.remotely.downloadSmall(bytecount=100*1024):
    ...
```

In this case, objects larger than `bytecount` are left in the cluster and returned as proxies, while smaller objects are downloaded and copied into the local scope.

Pure Python

The pyfora runtime supports a restricted, "purely functional" subset of python that we call "Pure Python". By "purely functional" we mean code in which:

- All data-structures are immutable (e.g. no modification of lists).
- No operations have side-effects (e.g. no sockets, no `print`).
- All operations are deterministic - running them on the same input always yields the same result (e.g. no access to system time, amount of available memory, etc.)

These restrictions are essential to the kinds of reasoning that pyfora applies to your code. Some of these restrictions may be relaxed in the future under certain circumstances, but at this time the following constraints are enforced:

- **Objects are immutable** (except for `self` within an object's `__init__()`). Expressions like `obj.x = 10` are disallowed, as they would modify `obj`. The exception to this rule is `self` within `__init__()`, where assignments are used to provide initial values to object members.
- **Lists are immutable**. Expressions like `a[0] = 10` won't work, nor will `a.append(10)`.

  However, given a list `a`, "appending" a value `x` to it by producing a new list using `a + [x]` results in effecient code without superflous copying of data.
- **Dictionaries are immutable**. In the future, updates to dictionaries will be allowed in cases where pyfora can prove that there is exactly one reference to the dictionary. But for the moment dictionaries can only be constructed from iterators, as in:

```
    dict((x, x**2) for x in xrange(100))
```

  Also note that at the moment, dictionaries can be quite slow, so use them sparingly.
- **No augmented assignment**. Expressions like `x += 10` are disallowed since they modify `x`.
- `print` is disabled.

- `locals()` and `globals()` are disabled.
- `del` is disabled.
- No `eval` or `exec`.

Note: Regular variable assignments **do** work as expected. The following code, for example, is allowed:

```
x = x + 5
v = [x]
v = v + [12]
```

Violation of Constraints

Whenever you invoke pyfora on a block of python code, the runtime attempts to give you either (a) the exact same answer you would have received had you run the same code in your python interpreter locally, or (b) an exception [1].

Constraint checking happens in two places. Some of the constraints are enforced at parse-time. As soon as you enter a `with executor.remotely` block, pyfora tries to determine all the code your invocation can touch. If any of that code contains syntatic elements that pyfora knows are invalid (such as `print()` statements), it will generate an exception.

Other constraints are enforced at runtime. For instance, the append method of lists, when invoked in pyfora, raises a *`InvalidPyforaOperation`* exception that's not catchable by python code running inside of pyfora. This indicates that the program has attempted to execute semantics that pyfora can't faithfully reproduce.

1.3.5 Performance Guide

Compilation and Parallelism

The pyfora runtime performs two kinds of optimization: JIT compilation, which ensures that single-threaded code is fast, and automatic parallelization, which ensures that you can use many cores at once. In both cases, the goal is to get as close as possible to the performance one can achieve using C with hand-crafted parallelism. However, as with most programming models, there are multiple ways to write the same program and these may have different performance characteristics. This section will help you understand what the pyfora VM can optimize and what it can't.

Generally speaking, there's a lot of overhead for invoking pyfora, since it has to compile your code, and has overhead shipping objects to the server. Don't expect pyfora to speed up things that are already pretty fast (i.e. less than a second or two).

For achieving maximum single-threaded code, the main takeaways are:

- Numerical calculations involving ints and floats are super fast - very close to what you can get with C.
- There is no penalty for using higher-order functions, `yield`, classes, etc. pyfora can usually optimize it all away.
- Repeatedly assigning to a variable in a way that causes it to assume many different types will caues a slow-down in code compilation. Avoid repeatedly assigning different types to one variable in a loop. Similarly, `for x in someList:` might be slower if the types in someList are heterogeneous.
- Tuples with a structure that is stable throughout your program (e.g. they always have three elements) will be very fast.

[1] Currently, the only intended exception to this rule is integer arithmetic: on the occurrence of an integer arithmetic overflow, pyfora will give you the semantics of the underyling hardware, whereas python will produce an object of type `long` with the correct value. Eventually, we will make this tradeoff configurable, but it has pretty serious performance implications, so for the moment we're just ignoring this difference.

- Tuples where the number of elements varies with program data will be slow - use lists for this.
- Lists prefer to be homogenously typed - e.g. a list with only floats in it will be faster to access than a list with floats and ints.
- There is more overhead for using a list in pyfora than in CPython - prefer a few large lists to a lot of small ones. [1]
- Deeply nested lists of lists may be slow. [1]
- Dictionaries are slow. [1]
- Strings are fast.

For achieving maximum parallelism, know these principles:

- pyfora parallelizes within stackframes - if you write `f(g(), h())`, pyfora will be able to run g() and h() in parallel.
- Parallel calls within parallel calls work.
- List comprehensions, `xrange()`, `sum()`, are parallel out of the box.
- pyfora parallelizes adaptively - it won't be triggered if all the cores in the system are saturated.
- pyfora won't currently parallelize `for` and `while` loops.
- Passing generator expressions into `sum()` or other parallelizable algorithms parallelizes.
- Large lists have a strong performance preference for "cache local" access. Code that touches neighboring list elements outperforms code that randomly accesses elements all over the list.

The pyfora JIT Compiler

Basic Behavior of the JIT

The pyfora JIT compiler optimizes the code your program spends the most time in. So, for instance, if you write:

```
def loopSum(x):
    result = 0.0
    while x > 0:
        result = result + x
        x = x - 1
    return result
print loopSum(1000000000)
```

The pyfora runtime will notice that your program is spending a huge amount of time in the while loop and produce a fast machine-code version of it in which x and result are held in registers (as 64 bit integer and floats, respectively). We generate machine code using the excellent and widely-used llvm project. In simple numerical programs, you'll end up with the same code you'd get from a good C++ compiler. Today, these are table-stakes for all JIT compilers.

Higher-Order Functions, Classes, and other Language Constructs

Unlike most JIT compilers applied to dynamically typed languages, pyfora is designed to work well with higher-order functions and classes. In general, this is a thorny problem for any system attempting to speed up python because in regular python programs, it's possible to modify class and instance methods during the execution of the program. This means that any generated code in tight loops has to repeatedly check to see whether some method call has changed[#mutating_code]_. Because this is disabled in pyfora code, pyfora can agressively optimize away these

[1] We consider this to be a performance defect that we can eventually fix. However, some of these defects will be easier to fix than others.

checks, perform agressive function inlining, and generally perform a lot of the optimizations you see in compilers optimizing statically typed languages like C++ or Java. This allows you to refactor your code into classes and objects without paying a performance penalty.

This flexibility comes at a cost: the compiler generates new code for every combination of types and functions that the it sees. For instance, if you write:

```
def loopSum(x, f):
    result = 0
    while x>0:
        result = result + f(x)
        x = x - 1
    return result
```

then the compiler will generate completely different code for `loopSum(1000000, lambda x:x)` and `loopSum(1000000, lambda x:x*x)` and both will be very fast. This extends to classes. For instance, if you write:

```
class Add:
    def __init__(self, arg):
        self.arg = arg

    def __call__(self, x):
        return self.arg + x

class Multiply:
    def __init__(self, arg):
        self.arg = arg

    def __call__(self, x):
        return self.arg * x

class Compose:
    def __init__(self, f, g):
        self.f = f
        self.g = g

    def __call__(self, x):
        return self.f(self.g(x))
```

you will get idential performance if you write `loopSum(1000000, lambda x:  x * 10.0 + 20.0)` and `loopSum(1000000, Compose(Multiply(10.0), Add(20.0)))` - they boil down to the same mathematical operations, and because pyfora doesn't allow class methods to be modified, it can reason about the code well enough to produce fast machine-code.

Keep the Total Number of Type Combinations Small

The pyfora compiler operates by tracking all the distinct combinations of types it sees for all the variables in a given stackframe, and generating code for each combination. This means that a function like:

```
def typeExplosion(v):
    a = None
    b = None
    c = None
    for x in v:
        if x < 1:
            a = x
        elif x < 2:
```

```
            b = x
        else:
            c = x
    return (a,b,c)
```

could generate a lot of code. For instance, if `a`, `b`, and `c` can all be None or an integer, you'll end up with 8 copies of the loop. That by itself isn't a problem, but if you keep adding variables, the total number of types grows exponentially - eventually, you'll wind up waiting forever for the compiler to finish generating code.

Tuples as Structure

Speaking of "types", pyfora considers function instances, class instances, and tuples to be "structural". This means that the compiler will agressively track type information about the contents of these objects. So, for instance, `lambda x: x + y` is a different type if `y` is an integer or a float in the surrounding scope. Similarly, `(x, y)` tracks the type information of both `x` and `y`. This is one of the reasons why there is no penalty for putting values into objects or tuples - the compiler tracks that type information the whole way through, so that `(x, y)[0]` is semantically equivalent to `x` regardless of what `y` is.

This is great until you start using tuples to represent data with a huge variety of structure, which can overwhelm the compiler. For instance,

```
def buildTuple(ct):
    res = ()
    for ix in xrange(ct):
        res = res + (ix,)
    return res
print buildTuple(100)
```

will produce a lot of code, because it will produce separate copies of the loop for the types "empty tuple", "tuple of one integer", "tuple of two integers", ..., "tuple of 99 integers", etc.

Because of this, tuples should generally be used when their shape will be stable (i.e. producing a small number of types) over the course of the program and you want the compiler to be able to see it. [3]

Also note that this means that if you have tuples with heterogeneous types and you index into it with a non-constant index, you will generate slower code. This is because the compiler needs to generate a separate pathway for each possible resulting type. For instance, if you write:

```
aTuple = (0.0, 1, "a string", lambda x: x)
functionCount = floatCount = 0

for ix in range(100):
    # pull an element out of the tuple - the compiler can't tell what
    # kind of element it is ahead of time
    val = aTuple[ix % len(aTuple)]

    if isinstance(val, type(lambda: None)):
        functionCount = functionCount + 1
    elif isinstance(val, float):
        floatCount = floatCount + 1
```

then the compiler will need to generate branch code at the `aTuple[...]` instruction. This will work, but will be slower than it would be if the tuple index could be known ahead of time.

[3] We expect to be able to fix this over the long run by identifying cases where we have an inordinate number of types and moving to a collapsed representation in which we don't track all the possible combinations.

Lists

Lists are designed to hold data that varies in structure. The compiler doesn't attempt to track the types of the individual objects inside of a list. Specifically, that means that `[1, 2.0]` and `[2.0, 1]` have the same type - they're both 'list of int and float', whereas `(1, 2.0)` and `(2.0, 1)` are different types.

Lists are fastest when they're homogenous (e.g. entirely containing elements of the same type). This is because the pyfora VM can pack data elements very tightly (since they all have the same layout) and can generate very efficient lookup code. Lists with heterogenous types are still fast, but the more types there are, the more code the compiler needs to generate in order to work with them, so try to keep the total number of types small.

In general, lists have more overhead in than in CPython [4] . This is because lists are the primary "big data" structure for pyfora - a list can be enormous (up to terabytes of data), and the data structure that represents them is rather large and complex. So, if possible, try to structure your program so that you create a few bigger lists, rather than a lot of little lists.

One exception to this rule: if `v` is a list, the operation: `v + [element]` will be fast and pyfora will optimize away the creation of the intermediate list and be careful not to duplicate `v` unless absolutely necessary. This is the fastest way to build a list.

Large lists are cheap to concatenate - they're held as a tree structure, so you don't have to worry that each time you concatenate you're making a copy.

Finally, avoid nesting lists deeply - this places a huge strain on the "big data" component of pyfora's internal infrastructure.

Dictionaries and Strings

Dictionaries are currently very slow [5] . Don't use them inside of loops.

Strings are fast. The pyfora string structure is 32 bytes, allowing the VM to pack any string of 30 characters or less into a data structure that doesn't hit the memory manager. Indexing into strings is also fast. Strings may be as large as you like (if necessary, they'll be split across machines).

Note that for strings that are under 100000 characters, string concatenation makes a copy, so you can accidentally get $O(N^2)$ performance behavior if you write code where you are repeatedly concatenating a large string to a small string.

Parallelism

The Core Model of Parallelism in pyfora

pyfora exploits "dataflow" parallelism at the stack-frame level. It operates by executing your code on a single thread and then periodically interrupting it and walking its stack, looking at the flow of data within each stack frame to see whether there are upcoming calls to functions that it can schedule while the current call is executing.

For instance, if you write `f(g(),h())`, then while executing the call to `g()`, the runtime can see that you are going to execute `h()` next. If you have unsaturated cores, it will rewrite the stack frame to also call `h()` in parallel. When both calls return, it will resume and call `f()`. You can think of this as fork-join parallelism where the forks are placed automatically.

As an example, the simple divide-and-conquer implementation of a `sum()` function could be written as:

[4] Another performance optimization we plan for the future will be to recognize the difference between small and large lists, and generate a faster implementation when we recognize ahead of time that lists are going to be small.

[5] Something we can fix, but not currently scheduled. Let us know if you need this.

```
def sum(a,b,f):
    if a >= b:
        return 0
    if a + 1 >= b:
        return f(a)

    mid = (a+b)/2

    return sum(a,mid,f) + sum(mid,b,f)
```

We can then write `sum(0, 1000000000000, lambda x: x**0.5)` and get a parallel implementation. This works because each call to sum contains two recursive calls to sum, and pyfora can see that these are independent.

Note that pyfora assumes that exceptions are rare - in the case of `f(g(),h())`, pyfora assumes that by default, `g()` is not going to throw an exception and that it can start working on `h()`. In the case where `g()` routinely throws exceptions, pyfora will start working on `h()` only to find that the work is not relevant. Some python idioms use exceptions for flow control: for instance, accessing an attribute and then catching `AttributeError` as a way of determining if an object meets an interface. In this case, make sure that you don't have an expensive operation in between the attribute check and the catch block.

Nested Parallelism

This model of parallelism naturally allows for nested parallelism. For instance, `sum(0,1000,lambda x: sum(0,x,lambda y:x*y))` will be parallel in the outer `sum()` but also in the inner `sum()`. This is because pyfora doesn't really distinguish between the two - it parallelizes stackframes, not algorithms.

Adaptive Parallelism

pyfora's parallelism is adaptive and dynamic - it doesn't know ahead of time how the workload is distributed across your functions. It operates by agressively splitting stackframes until cores are saturated, waiting for threads to finish, and then splitting additional threads.

This model is particularly effective when your functions have different runtimes depending on their input. For instance, consider:

```
def isPrime(p):
    if p < 2: return 0
    x = 2
    while x*x <= p:
        if p%x == 0:
            return 1
        x = x + 1
    return 1

sum(isPrime(x) for x in xrange(10000000))
```

Calls to `isPrime()` with large integers take a lot longer than calls to `isPrime()` with small integers, because we have to divide so many more numbers into the large ones. Naively allocating chunks of the 10000000 range to cores will end up with some cores working while others finish their tasks early. pyfora can handle this because it sees the fine structure of parallelism available to sum and can repeatedly subdivide the larger ranges, keeping all the cores busy.

This technique works best when your tasks subdivide to a very granular level. In the case where you have a few subtasks with long sections of naturally single-threaded code, pyfora may not schedule those sections until partway through the calculation. You'll get better performance if you can find a way to get the calculation to break down as finely as possible.

It's also important to note that the pyfora VM doesn't penalize you for writing your code in a parallel way. pyfora machine-code is optimized for single-threaded execution - it's only when there are unused cores and pyfora wants more tasks to work on that we split stackframes, in which case we pop the given stackframe out of native code and back into the interpreter.

The one caveat here is that function calls have stack-frame overhead. Code that's optimized for maximum performance sometimes has conditions to switch it out of a recursive "parallelizable" form and into a loop. This is a tradeoff between single-threaded performance and parallelism granularity.

List Comprehensions and Sum are Parallel

By default, list comprehensions like `[isPrime(x) for x in xrange(10000000)]` are parallel if the generator in the righthand side supports the `__pyfora_generator__` parallelism model, which both `xrange()`, and lists support out of the box.

Similarly, functions like `sum()` are parallel if their argument supports the `__pyfora_generator__` interface. Note that this subtly changes the semantics of `sum:()` in standard python, `sum(f(x) for x in xrange(4))` would be equivalent to:

```
(((f(0)+f(1))+f(2))+f(3))+f(4)
```

performing the addition operations linearly from left to right. In the parallel case, we have a tree structure:

```
(f(0)+f(1)) + (f(2)+f(3))
```

when addition is associative. Usually this produces the same results, but it's not always true. For instance, roundoff errors in floating point arithmetic mean that floating point addition is not perfectly associative [6] . As this is a deviation from standard python, we plan to make it an optional feature in the future.

Loops are Sequential

Note that pyfora doesn't try to parallelize across loops. The `isPrime()` example above runs sequentially. In the future, we plan to implement loop unrolling so that if you write something like:

```
res = None
for x in xrange(...):
    res = f(g(x), res)
```

if the calls to `g()` are sufficiently expensive, we'll be able to schedule those calls in parallel and then execute the naturally sequential calls to `f()` as they complete. For the moment, however, assume that while and for loops are sequential (although functions inside them are all candidates for parallelism).

Lists Prefer Cache-Local Access

Lists are the basic building-block for "big data" in pyfora. A list that's large enough will get split across multiple machines. pyfora organizes a list's data into chunks of contiguous indices, where each chunk represents ~50-100 MB of data.

When one of your threads (in this context, a thread is just a collection of stackframes of python code that pyfora hasn't decided to subdivide) indexes into a very large list and that data isn't on the same machine as the thread, pyfora must decide what to do: (a) move the thread to the data, or (b) move the data to the thread? This is called a "cache miss." Threads tend to be much smaller than 50MB, so usually it will move the thread to the remote machine.

[6] For instance, `10e30 + (-10e30) + 10e-50` is not the same as `10e30 + ((-10e30) + 10e-50)`

One of the unique characteristics of the pyfora runtime: it will simulate the execution of code in advance of its execution to predict cache misses and move data and threads accordingly. For example, if your thread starts accessing two different blocks in a list, and those two blocks are on different machines, that thread may end up bouncing back and forth between the two machines in a slow oscillatory pattern. pyfora can predict these access patterns and optimize the layout of blocks and threads to prevent this in advance.

All of this infrastructure is useless if you index randomly into very big lists (here, we mean bigger than ~25% of a machine's worth of data). This is because it's now impossible for the scheduler to find an allocation of blocks to machines where a large fraction of your list accesses don't require you to cross a machine boundary.

As a result, you'll get the best performance if you can organize your program so that list accesses are "cache local", meaning that when you access one part of a list you tend to access other parts of the list that are nearby in the index space [7] .

Footnote

1.3.6 Working With Data in S3

Amazon's Simple Storage Service (S3) is a highly scalable, durable, general purpose store, that has been around since the original launch of Amazon Web Services (AWS), and is one of their most widely used services.

Whether you run a pyfora cluster in AWS or locally, pyfora lets you work with datasets stored in S3 in much the same way you would use files on your local disk.

Reading From S3

pyfora lets you treat files stored in S3 as if they are regular python strings even if they are much larger than amount of memory available on any machine in your cluster. The *importS3Dataset()* function creates a *RemotePythonObject* that represents the entire content of the specified file in S3 as a string of bytes, which can then be parsed into different data-structures.

For example, to parse a CSV file in S3 into a `pandas.DataFrame`:

```
import pyfora
import pyfora.pandas_util

executor = pyfora.connect('http://<cluster_manager_address>:30000')

data_as_string = executor.importS3Dataset('bucket_name', 'path/to/file.csv')
with executor.remotely:
    data_frame = pyfora.pandas_util.read_csv_from_string(data_as_string)

    # data_frame is a pandas.DataFrame that lives in memory in the pyfora cluster
    num_of_rows = len(data_frame)

    # do stuff with data_frame...

print "Num of rows:", num_of_rows.toLocal().result()
```

[7] In the future, we plan to implement a "streaming read" model for inherently non-cache-local algorithms. Essentially the idea is to use the same simulation technique that we use to determine what your cache misses are going to be, but instead of using them for scheduling purposes, we will actually fetch the values and merge them back into the program. In a good implementation, this should allow for a very low per-value overhead scattered value read.

Writing to S3

exportS3Dataset() is used to write strings into S3. For example:

```
import pyfora

executor = pyfora.connect('http://<cluster_manager_address>:30000')

with executor.remotely:
    large_string = 'lots of data ' * 10**9

executor.exportS3Dataset(large_string, 'bucket_name', 'path/to/file.txt')
```

AWS Credentials

To access private data in S3, the pyfora cluster must be given credentials with appropriate read and/or write permissions to the buckets and keys being used. The pyfora worker service reads AWS credentials from two environment variables: `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. These are the same variables used by `boto` and the AWS CLI tools.

When launching pyfora services in docker containers, you can set these variables as part of the `docker run` command. For example:

```
docker run -d -e AWS_ACCESS_KEY_ID=<key> -e AWS_SECRET_ACCESS_KEY=<secret> ufora/service
```

1.3.7 Linear Regression

This tutorial demonstrates using pyfora to:

1. Load a large CSV file from Amazon S3
2. Parse it into a `pandas.DataFrame`
3. Run linear regression on the loaded DataFrame
4. Download the regression coefficients and intercept back to python

---

Important: The example below uses a **large** dataset. It is a 64GB csv file that parses into 20GB of normally-distributed, randomly generated floating point numbers. It takes about 10 minutes to run on three c3.8xlarge instances in EC2.

You can use the `pyfora_aws` script installed with the pyfora package to easily set up a pyfora cluster in EC2 using either on-demand or spot instances.

If you prefer to try a (much) smaller version of this example, you can use the 5.2GB dataset `iid-normal-floats-13mm-by-17.csv`, by modifying line 9 below accordingly.

```
1 import pyfora
2 from pyfora.pandas_util import read_csv_from_string
3 from pyfora.algorithms import linearRegression
4
5 print "Connecting..."
6 executor = pyfora.connect('http://<cluster_manager>:30000')
7 print "Importing data..."
8 raw_data = executor.importS3Dataset('ufora-test-data',
9                                      'iid-normal-floats-20GB-20-columns.csv').result()
```

```
10
11 print "Parsing and regressing..."
12 with executor.remotely:
13     data_frame = read_csv_from_string(raw_data)
14     predictors = data_frame.iloc[:, :-1]
15     responses = data_frame.iloc[:, -1:]
16
17     regression_result = linearRegression(predictors, responses)
18     coefficients = regression_result[:-1]
19     intercept = regression_result[-1]
20
21
22 print 'coefficients:', coefficients.toLocal().result()
23 print 'intercept:', intercept.toLocal().result()
```

If you are familiar with `pandas` the code above should look quite familiar. After connecting to a pyfora cluster using *`pyfora.connect()`* in line 6, we import a dataset from Amazon S3 in line 8 using *`importS3Dataset()`*.

The value `raw_data` returned from *`importS3Dataset()`* is a *`RemotePythonObject`* that represents the entire dataset as a string. The data itself is lazily loaded to memory in the cluster when it is needed.

All the code inside the `with executor.remotely:` block that starts in line 12 is shipped to the cluster and executes remotely.

We use *`read_csv_from_string()`* to read the CSV in `raw_data` and produce a DataFrame.

Our regression fits a linear model to predict the last column from the prior ones. The *`linearRegression()`* algorithm is used to return an array with the linear model's coefficients and intercept.

In lines 22 and 23, outside the `with executor.remotely:` block, we bring some of the values computed remotely back into the local python environment. Values assigned to variables inside the `with executor.remotely:` are left in the pyfora cluster by default because they can be very large - much larger than the amount of memory available on your machine. Instead, they are represented locally using *`RemotePythonObject`* instances that can be downloaded using their *`toLocal()`* function.

1.3.8 pyfora_aws: Run pyfora on AWS

`pyfora_aws` is a command-line tool that makes it easy to launch and manage pyfora compute clusters on AWS. It is installed as part of the pyfora package.

Note: All instances in a cluster run in the same EC2 region, VPC and subnet (if using VPC), and security group. If you need to run more than one cluster in a region, use different VPCs, subnets, or security groups.

---

start

Launches one or more backend instances.

```
Usage: pyfora_aws start [OPTIONS]

Optional arguments:
-h, --help                  show this help message and exit
-y, --yes-all               Do not prompt user input. Answer "yes" to all prompts.

--ec2-region EC2_REGION
                            The EC2 region in which instances are launched. Can
```

```
                            also be set using the PYFORA_AWS_EC2_REGION
                            environment variable. Default: us-east-1

--vpc-id VPC_ID
                            The id of the VPC into which instances are launched.
                            EC2 Classic is used if this argument is omitted.

--subnet-id SUBNET_ID
                            The id of the VPC subnet into which instances are launched.
                            This argument must be specified if --vpc-id is used
                            and is ignored otherwise.

--security-group-id SECURITY_GROUP_ID
                            The id of the EC2 security group into which instances are launched.
                            If omitted, a security group called "pyfora ssh" (or "pyfora open"
                            if --open-public-port is specified) is created. If a security group
                            with that name already exists, it is used as-is.

-n NUM_INSTANCES, --num-instances NUM_INSTANCES
                            The number of instances to launch. Default: 1

--ssh-keyname SSH_KEYNAME
                            The name of the EC2 key-pair to use when launching instances.
                            Can also be set using the PYFORA_AWS_SSH_KEYNAME environment variable.

--spot-price SPOT_PRICE
                            Launch spot instances with specified max bid price.
                            On-demand instances are launch if this argument is omitted.

--instance-type INSTANCE_TYPE
                            The EC2 instance type to launch.
                            Default: c3.8xlarge

--open-public-port
                            If specified, HTTP access to the manager machine will
                            be open from anywhere (0.0.0.0/0). Use with care!
                            Anyone will be able to connect to your cluster. As an
                            alternative, considering tunneling pyfora's HTTP port
                            (30000) over SSH using the -L argument to the `ssh` command.

--commit COMMIT
                            Run the backend services from a specified commit in the ufora/ufora
                            GitHub repository.
```

Examples

```
$ pyfora_aws start --vpc-id vpc-0c73f14e --subnet-id subnet-7214f1a0 --ssh-keyname my_key -n 3
```

This will launch a cluster of three c3.8xlarge instances into the specified VPC and subnet in the default us-east-1 region, and use the EC2 ssh key-pair called `my_key`.

```
$ pyfora_aws start --instance-type g2.2xlarge --spot-price 0.3 --open-public-port
```

This will launch a single g2.2xlarge spot instance with a maximum bid price of $0.3 and open inbound traffic on port 30000.

add

Adds one or more workers to a running cluster.

```
Usage: pyfora_aws add [OPTIONS]

optional arguments:
-h, --help                    show this help message and exit

--ec2-region EC2_REGION
                              The EC2 region in which instances are launched. Can
                              also be set using the PYFORA_AWS_EC2_REGION
                              environment variable. Default: us-east-1

--vpc-id VPC_ID               The id of the VPC into which instances are launched.
                              EC2 Classic is used if this argument is omitted.

--subnet-id SUBNET_ID
                              The id of the VPC subnet into which instances are
                              launched. This argument must be specified if --vpc-id
                              is used and is ignored otherwise.

--security-group-id SECURITY_GROUP_ID
                              The id of the EC2 security group into which instances
                              are launched.

-n NUM_INSTANCES, --num-instances NUM_INSTANCES
                              The number of instances to launch. Default: 1

--spot-price SPOT_PRICE
                              Launch spot instances with specified max bid price.
                              On-demand instances are launch if this argument is
                              omitted.
```

Note: Instance type is selected automatically based on the type of instances already running. It is not possible to mix different types of instances in the same cluster.

Examples

```
$ pyfora_aws add -n 3 --ec2-region us-west-2 --security-group-id sg-2f28a1c0
```

This adds three instances to an existing cluster running in the `us-west-2` region with security group `sg-2f28a1c0`.

list

Print a list of running backend instances.

```
usage: pyfora_aws list [OPTIONS]

optional arguments:
-h, --help                    show this help message and exit

--ec2-region EC2_REGION
```

```
                            The EC2 region in which instances are launched. Can
                            also be set using the PYFORA_AWS_EC2_REGION
                            environment variable. Default: us-east-1

--vpc-id VPC_ID             The id of the VPC into which instances are launched.
                            EC2 Classic is used if this argument is omitted.

--subnet-id SUBNET_ID
                            The id of the VPC subnet into which instances are
                            launched. This argument must be specified if --vpc-id
                            is used and is ignored otherwise.

--security-group-id SECURITY_GROUP_ID
                            The id of the EC2 security group into which instances
                            are launched. If omitted, a security group called
                            "pyfora ssh" (or "pyfora open" if --open-public-port
                            is specified) is created. If a security group with
                            that name already exists, it is used as-is.
```

Examples

```
 $ pyfora_aws list --ec2-region us-west-1
 3 instances:
     i-dc7acd1f | 50.18.72.241 | running | worker
     i-387ccbfb | 54.176.35.132 | running | worker
     i-ba7bcc79 | 54.177.18.215 | running | worker
```

stop

Stops all backend instances in the specified region, VPC and subnet, and security-group.

```
Usage: pyfora_aws stop [OPTIONS]

optional arguments:
-h, --help                  show this help message and exit

--ec2-region EC2_REGION
                            The EC2 region in which instances are launched. Can
                            also be set using the PYFORA_AWS_EC2_REGION
                            environment variable. Default: us-east-1

--vpc-id VPC_ID             The id of the VPC into which instances are launched.
                            EC2 Classic is used if this argument is omitted.

--subnet-id SUBNET_ID
                            The id of the VPC subnet into which instances are
                            launched. This argument must be specified if --vpc-id
                            is used and is ignored otherwise.

--security-group-id SECURITY_GROUP_ID
                            The id of the EC2 security group into which instances
                            are launched. If omitted, a security group called
                            "pyfora ssh" (or "pyfora open" if --open-public-port
                            is specified) is created. If a security group with
                            that name already exists, it is used as-is.
```

```
--terminate                 Terminate running instances. Otherwise, they are just stopped.
```

Examples

```
 $ pyfora_aws stop --ec2-region us-west-1 --terminate
 Terminating 3 instances:
     i-dc7acd1f | 50.18.72.241 | running | worker
     i-387ccbfb | 54.176.35.132 | running | worker
     i-ba7bcc79 | 54.177.18.215 | running | worker
```

deploy

Deploys a build to all running instances.

Note: This command is typically only used during development of backend services. It is rarely used in normal operations.

```
Usage: pyfora_aws deploy -i IDENTITY_FILE -p PACKAGE [OPTIONS]

optional arguments:
-h, --help                  show this help message and exit

-i IDENTITY_FILE, --identity-file IDENTITY_FILE
                            The file from which the private SSH key is read.

-p PACKAGE, --package PACKAGE
                            Path to the backend package to deploy.

--ec2-region EC2_REGION
                            The EC2 region in which instances are launched. Can
                            also be set using the PYFORA_AWS_EC2_REGION
                            environment variable. Default: us-east-1

--vpc-id VPC_ID             The id of the VPC into which instances are launched.
                            EC2 Classic is used if this argument is omitted.

--subnet-id SUBNET_ID
                            The id of the VPC subnet into which instances are
                            launched. This argument must be specified if --vpc-id
                            is used and is ignored otherwise.

--security-group-id SECURITY_GROUP_ID
                            The id of the EC2 security group into which instances
                            are launched. If omitted, a security group called
                            "pyfora ssh" (or "pyfora open" if --open-public-port
                            is specified) is created. If a security group with
                            that name already exists, it is used as-is.
```

1.3.9 pyfora

pyfora.**connect**(*url*, *timeout=30.0*)
Opens a connection to a pyfora cluster

Parameters

- **url** (*str*) – The HTTP URL of the cluster's manager (e.g. `http://192.168.1.200:30000`)
- **timeout** (*Optional float*) – A timeout for the operation in seconds, or None to wait indefinitely.

Returns An *Executor* that can be used to submit work to the cluster.

Exceptions

exception pyfora.**PyforaError**
Base class for all pyfora exceptions.

exception pyfora.**ConnectionError**
Raised when a connection to the pyfora backend cannot be established.

exception pyfora.**NotCallableError**
Raised when an attempt is made to call a non-callable object.

exception pyfora.**ComputationError**(*remoteException*, *trace*)
Raised when a remote computation results in an exception.

Parameters

- **remoteException** (*Exception*) – The exception raised by the remote computation.
- **trace** (*Optional[List]*) – A representation of the stack trace in which the exception was raised. It takes the form: `[{'path':str, 'line': int}, ... ]`

exception pyfora.**PythonToForaConversionError**(*message*, *trace=None*)
Raised when an attempt is made to use a Python object that cannot be remoted by `pyfora`.

This may happen when, for example:

•A function attempts to mutate state or produce side-effect (i.e. it is not "purely functional").

•A call is made to a Python builtin that is not supported by `pyfora` (e.g. `open()`)

Parameters

- **message** (*str*) – Error message.
- **trace** (*Optional[List]*) – A representation of the stack trace in which the exception was raised. It takes the form: `[{'path':str, 'line': int}, ... ]`

exception pyfora.**ForaToPythonConversionError**
Raised when attempting to download a remote object that cannot be converted to Python.

exception pyfora.**PyforaNotImplementedError**
Feature not yet implemented in `pyfora`.

exception pyfora.**InvalidPyforaOperation**
Raised when a running computation performs an operation that cannot be faithfully executed with `pyfora`.

exception pyfora.**ResultExceededBytecountThreshold**
Raised when attempting to download a remote object whose size exceeds the specified maximum.

Executor

class pyfora.Executor.**Executor**(*connection*, *pureImplementationMappings=None*)
Submits computations to a pyfora cluster and marshals data to/from the local Python.

The Executor is the main point of interaction with a pyfora cluster. It is responible for sending computations to the cluster and returning the result as a RemotePythonObject future.

It is modeled after the same-named abstraction in the concurrent.futures module that is part of the Python3 standard library.

All interactions with the remote cluster are asynchronous and return *Future* objects that represent the in-progress operation.

Python objects are sent to the server using the *define()* method, which returns a *Future* that resolves to a *RemotePythonObject* corresponding to the submitted object.

Similarly, functions and their arguments can be submitted using the *submit()* method which returns a *Future* that resolves to a *RemotePythonObject* of the evaluated expression or raised exception.

Note: This class is not intended to be constructed explicitly. Instances of it are created by calling *connect()*.

Parameters

- **connection** (*pyfora.Connection.Connection*) – an open connection to a cluster.
- **pureImplementationMappings** (*optional*) – a PureImplementationMappings that defines mapping between Python libraries and their "pure" *pyfora* implementation.

close()
Closes the connection to the pyfora cluster.

define(*obj*)
Create a remote representation of an object.

Sends the specified object to the server and return a Future that resolves to a RemotePythonObject representing the object on the server.

Parameters **obj** – A python object to send

Returns A *Future* that resolves to a *RemotePythonObject* representing the object on the server.

exportS3Dataset(*valueAsString*, *bucketname*, *keyname*)
Write a ComputedRemotePythonObject representing a *pyfora* string to S3

Parameters

- **valueAsString** (RemotePythonObject.ComputedRemotePythonObject) – a computed string.
- **bucketname** (*str*) – The name of the S3 bucket to write to.
- **keyname** (*str*) – The S3 key to write to.

Returns A *Future* representing the completion of the export operation. It resolves either to None (success) or to an instance of *PyforaError*.

getWorkerCount()
Returns the number of workers connected to the cluster.

Returns The number of workers currently available in the cluster.

Return type int

importRemoteFile(*path*)
Loads the content of a file as a string

Note: The file must be available to all machines in the cluster using the specified path. If you run multiple workers you must either copy the file to all machines, or if using a network file-system, mount it into the same path on all machines.

In addition, pyfora may cache the content of the file. Changes to the file's content made after it has been loaded may have no effect.

Parameters **path** (*str*) – Full path to the file. This must be a valid path on **all** worker machines in the cluster.

Returns A *Future* that resolves to a *RemotePythonObject* representing the content of the file as a string.

importS3Dataset(*bucketname*, *keyname*, *verify=True*)
Creates a *RemotePythonObject* that represents the content of an S3 key as a string.

Parameters

- **bucketname** (*str*) – The S3 bucket to read from.
- **keyname** (*str*) – The S3 key to read.
- **verify** – Throw an exception immediately if the key or bucket cannot be read.

Returns A *Future* that resolves to a *RemotePythonObject* representing the content of the S3 key.

isClosed()
Determine if the *Executor* is connected to the cluster.

Returns `True` if *close()* has been called, `False` otherwise.

Return type bool

remotely
Returns a *WithBlockExecutor.WithBlockExecutor* that can be used to enter a block of "pure" Python code.

The `with executor.remotely:` syntax allows you to automatically submit an entire block of python code for remote execution. All the code nested in the remotely `with` block is submitted.

Returns A *WithBlockExecutor* that extracts python code from a with block and submits it to the pyfora cluster for remote execution. Results of the remote execution are returned as RemotePythonObject and are automatically reasigned to their corresponding local variables in the with block.

submit(*fn*, **args*)
Submits a callable to be executed on the cluster with the provided arguments.

This function is shorthand for calling *define()* on the callable and all arguments and then invoking the remote callable with the remoted arguments.

Returns A *Future* representing the given call. The future eventually resolves to a *RemotePythonObject* instance or an exception.

WithBlockExecutor

class pyfora.WithBlockExecutor.**WithBlockExecutor**(*executor*)
A helper object used to synchronously run blocks of code on a cluster.

When entering a `with` block using a *WithBlockExecutor*, the body of the block is extracted and submitted to the pyfora cluster for execution, along with all its local dependencies. Variable assignments within the block are returned as *RemotePythonObject* and reassigned to their corresponding local varialbes when exiting the block.

Use *downloadAll()*, *remoteAll()*, and *downloadSmall()* to modify the default behavior and select which objects should be downloaded from the server and which objects should be returned as *RemotePythonObject* futures.

Note: Instances of *WithBlockExecutor* are only intended to be created by *Executor*. User code typically uses *remotely* to get a *WithBlockExecutor*.

---

downloadAll()
Modify the executor to download all results into the local namespace.

Returns `self` to support chaining.

downloadSmall(*bytecount=10000*)
Modify the executor to download small results into the local namespace and return proxies for everything else.

Returns `self` to support chaining.

remoteAll()
Modify the executor to leave all results on the server and only return proxies (default).

Returns `self` to support chaining.

withStatusCallback(*callback*)
Modify the executor to call 'callback' while computations are blocked with status updates.

'callback' will receive a json package from the server containing information about the current computation. This will override the default callback, which attempts to determine whether we're in a jupyter notebook.

RemotePythonObject

A proxy for some object, data or callable that lives in memory on a pyfora cluster

class pyfora.RemotePythonObject.**RemotePythonObject**(*executor*)
A local proxy for a python object that lives in memory on a pyfora cluster.

This is an abstract class and should not be used directly, but through its two subclasses: *DefinedRemotePythonObject* and *ComputedRemotePythonObject*.

Parameters **executor** – An *Executor*

toLocal()
Downloads the remote object.

Returns A *Future* that resolves to the python object that this *RemotePythonObject* represents.

RemotePythonObject.DefinedRemotePythonObject

class pyfora.RemotePythonObject.**DefinedRemotePythonObject**(*objectId*, *executor*)
A proxy that represents a local object, which has been uploaded to a pyfora cluster.

Note: Only *Executor* is intended to create instances of *DefinedRemotePythonObject*. They are created by calling *define()*.

Parameters

- **objectId** (*int*) – a value that uniquely identifies the remote object that this *DefinedRemotePythonObject* represents.
- **executor** – the *Executor* that created this *DefinedRemotePythonObject*.

RemotePythonObject.ComputedRemotePythonObject

class pyfora.RemotePythonObject.**ComputedRemotePythonObject**(*computedValue*, *executor*, *isException*)
A proxy that represents a remote object created on a pyfora cluster as a result of some computation.

Note: Only *Executor* is intended to create instances of *ComputedRemotePythonObject*. They are created by calling *submit()*.

Parameters

- **computedValue** – an instance of a SubscribableWebObject computedValue representing the computation that produced this *ComputedRemotePythonObject*.
- **executor** – the *Executor* that created this *DefinedRemotePythonObject*.

Future

class pyfora.Future.**Future**(*onCancel=None*)
Bases: concurrent.futures._base.Future

This pyfora.Future object subclasses the standard Python concurrent.futures._base.Future object. See: http://pythonhosted.org/futures/ https://pypi.python.org/pypi/futures

Futures wrap the result to an asynchronous computation which can be accessed by a blocking call to `result()`.

The pyfora Future object extends the concurrent.futures object by supporting cancellation with the *cancel()* method.

cancel()
Cancel a running computation

Algorithms

Linear Regression

pyfora.algorithms.**linearRegression**(*predictors*, *responses*)

Compute the regression coefficients (with intercept) for a set of predictors against responses.

Parameters

- **predictors** (*DataFrame*) – a `pandas.DataFrame` with the predictor columns.
- **responses** (*DataFrame*) – a `pandas.DataFrame` whose first column is used as the regression's target.

Returns

A `numpy.array` with the regression coefficients. The last element in the array is the intercept.

Logistic Regression

class pyfora.algorithms.**BinaryLogisticRegressionFitter**(*C*, *hasIntercept=True*, *method='newton-cg'*, *interceptScale=1.0*, *tol=0.0001*, *maxIter=100000.0*, *splitLimit=1000000*)

A logistic regression "fitter" ithat holds fitting parameters used to fit logit models.

Parameters

- **C** (*float*) – Inverse of regularization strength; must be a positive float.
- **hasIntercept** (*bool*) – If True, include an intercept (aka bias) term in the fitted models.
- **method** (*string*) – one of 'newton-cg' (default) or 'majorization'
- **interceptScale** (*float*) – When `hasIntercept` is True, feature vectors become `[x, interceptScale]`, i.e. we add a "synthetic" feature with constant value `interceptScale` to all of the feature vectors. This synthetic feature is subject to regularization as all other features. To lessen the effect of regularization, users should increase this value.
- **tol** (*float*) – Tolerance for stopping criteria. Fitting stops when the l2-norm of the parameters to update do not change more than `tol`.
- **maxIter** (*int*) – A hard limit on the number of update cycles allowed.

fit(*X*, *y*)

fit a (regularized) logit model to the predictors `X` and responses `y`.

Parameters

- **X** – a dataframe of feature vectors.
- **y** – a dataframe (with one column) which contains the "target" values, corresponding to the feature vectors in `X`.

Returns A `BinaryLogisticRegressionModel` which represents the fit model.

Example:

```
        # fit a logit model without intercept using regularizer 1.0

        from pyfora.algorithms import BinaryLogisticRegressionFitter

        fitter = BinaryLogisticRegressionFitter(1.0, False)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})

        model = fitter.fit(x, y)
```

class pyfora.algorithms.logistic.BinaryLogisticRegressionModel.**BinaryLogisticRegressionModel**(

Represents a fit logit model.

coefficients
numpy.array – The regressions coefficients.

intercept
float – The fitted model's intercept

Note: This class is not intended to be constructed directly. Instances of it are returned by *fit()*.

---

predict(*X*)
Predict the class labels of X.

Parameters **X** (*DataFrame, or numpy.array*) – a set of feature vectors

Returns array containing the predicted class labels.

Return type numpy.array

predict_probability(*X*)
Estimate the conditional class-zero probability for the features in X.

Parameters **X** (*DataFrame, or numpy.array*) – a set of feature vectors

Returns array containing the predicted probabilities.

Return type numpy.array

score(*X*, *y*)
Returns the mean accuracy on the given test data and labels.

Parameters

- **X** (*DataFrame*) – Feature vectors

- **y** (*DataFrame*) – Target labels, corresponding to the vectors in X.

Returns The mean accuracy of *predict()* with respect to y.

Return type float

Regression Trees

class pyfora.algorithms.regressionTrees.RegressionTree.**RegressionTreeBuilder**(*maxDepth*, *minSamplesSplit=2*, *numBuckets=10000*, *minSplitThresh=1000000*)

Fits regression trees to data using specified tree parameters.

Parameters

- **maxDepth** (*int*) – The maximum depth of a fit tree
- **minSamplesSplit** (*int*) – The minimum number of samples required to split a node
- **numBuckets** (*int*) – The number of buckets used in the estimation of optimal column splits.
- **minSplitThresh** (*int*) – an "internal" argument, not generally of interest to casual users, giving the splitting rule in computeBucketedSampleSummaries.

Returns A *RegressionTree* instance.

Examples:

```
from pyfora.algorithms import RegressionTreeBuilder

builder = RegressionTreeBuilder(2)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
y = pandas.DataFrame({'y': [0,1,1]})
regressionTree = builder.fit(x, y)
```

static buildTree(*x*, *y*, *minSamplesSplit*, *maxDepth*)

Fit a regression tree to predictors *x* and responses *y* using parameters *minSamplesSplit* and *maxDepth*.

Parameters

- **x** (pandas.DataFrame) – of the predictors.
- **y** (pandas.DataFrame) – giving the responses.
- **maxDepth** – The maximum depth of a fit tree
- **minSamplesSplit** – The minimum number of samples required to split a node

fit(*x*, *y*)

Using a *RegressionTreeBuilder*, fit a regression tree to predictors *x* and responses *y*.

Parameters

- **x** (pandas.DataFrame) – of the predictors.
- **y** (pandas.DataFrame) – giving the responses.

Returns a *RegressionTree* instance.

Examples:

```
builder = pyfora.algorithms.regressionTrees.RegressionTree.RegressionTreeBuilder(2)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
y = pandas.DataFrame({'y': [0,1,1]})
regressionTree = builder.fit(x, y)
```

class pyfora.algorithms.regressionTrees.RegressionTree.**RegressionTree**(*rules*, *numDimensions=None*, *columnNames=None*)

A class representing a regression tree.

A regression tree is represented, essentially, as a list of "rules", which are either SplitRule, giving "split" nodes, which divide the domain by a hyperplane, or RegressionLeafRule, which just hold a prediction value.

Note: This class is not generally instantiated directly by users. Instead, they are normally returned by *RegressionTreeBuilder*.

---

predict(*x*, *depth=None*)

Predicts the responses corresponding to pandas.DataFrame x.

Returns A pandas.Series giving the predictions of the rows of x.

Examples:

```
from pyfora.algorithms import RegressionTreeBuilder

builder = RegressionTreeBuilder(2)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
y = pandas.DataFrame({'y': [0,1,1]})
regressionTree = builder.fit(x, y)

# predict `regressionTree` on `x` itself
regressionTree.predict(x)
```

score(*x*, *yTrue*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as `(1 - u / v)`, where `u` is the regression sum of squares `((yTrue - yPredicted) ** 2).sum()` and `v` is the residual sum of squares `((yTrue - yTrue.mean()) ** 2).sum()`. Best possible score is `1.0`, lower values are worse.

Returns (float) the R^2 value

Examples:

```
from pyfora.algorithms import RegressionTreeBuilder

builder = RegressionTreeBuilder(2)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
```

```
        y = pandas.DataFrame({'y': [0,1,1]})
        regressionTree = builder.fit(x, y)

        # predict `regressionTree` on `x` itself
        regressionTree.score(x, y)
```

Gradient Boosting

class pyfora.algorithms.regressionTrees.GradientBoostedRegressorBuilder.**GradientBoostedRegres**

A class which builds (or "fits") gradient-boosted regression trees to data with specified parameters. These parameters are

Parameters

- **maxDepth** (*int*) – The max depth allowed of each constituent regression tree.
- **nBoosts** (*int*) – The number of "boosting iterations" used.
- **learningRate** (*float*) – The learning rate of the model, used for regularization. Each successive tree from boosting stages are added with multiplier `learningRate`.
- **minSamplesSplit** (*int*) – The minimum number of samples required to split a regression tree node.
- **numBuckets** (*int*) – The number of buckets used in the estimation of optimal column splits for building regression trees.
- **loss** – the loss used when forming gradients. Defaults to `l2`, for least-squares loss. The only other allowed value currently is `lad`, for "least absolute deviation" (aka l1-loss).

fit (*X*, *y*)

Fits predictors `X` to responses `y`.

Parameters

- **X** (`pandas.DataFrame`) – predictors.
- **y** (`pandas.DataFrame`) – responses.

Returns A *RegressionModel* instance.

Examples:

```
        from pyfora.algorithms import GradientBoostedRegressorBuilder

        builder = GradientBoostedRegressorBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})
```

```
        model = builder.fit(x, y)
```

iterativeFitter(*X*, *y*)

Returns an *IterativeFitter* instance which can iteratively fit boosting models.

Parameters

- **X** (pandas.DataFrame) – predictors.
- **y** (pandas.DataFrame) – responses.

Examples:

```
        from pyfora.algorithms import GradientBoostedRegressorBuilder

        builder = GradientBoostedRegressorBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})

        fitter = builder.iterativeFitter(x, y)

        # compute scores vs number of boosts
        numBoosts = 5
        scores = []
        for ix in xrange(numBoosts):
            fitter = fitter.next()
            scores = scores + [fitter.model.score(x, y)]
```

class pyfora.algorithms.regressionTrees.RegressionModel.**RegressionModel**(*additiveRegressionTree*, *X*, *XDimensions*, *yAsSeries*, *loss*, *regressionTreeBuilder*, *learningRate*)

A class representing a gradient-boosted regression tree model fit to data.

Note: These classes are not normally instantiated directly. Instead, they are typically returned by *GradientBoostedRegressorBuilder* instances.

---

predict(*df*, *nEstimators=None*)

Predict on the pandas.DataFrame df.

Example:

```
        from pyfora.algorithms import GradientBoostedRegressorBuilder

        builder = GradientBoostedRegressorBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})
```

```
        model = builder.fit(x, y)

        # predict `x` using the model `model`:
        model.score(x, y)
```

score(*X*, *yTrue*)

Return the coefficient of determination (R^2) of the prediction.

The coefficient R^2 is defined as `(1 - u / v)`, where `u` is the regression sum of squares `((yTrue - yPredicted) ** 2).sum()` and `v` is the residual sum of squares `((yTrue - yTrue.mean()) ** 2).sum()`. Best possible score is `1.0`, lower values are worse.

Parameters

- **X** – the predictor DataFrame.
- **yTrue** – the (true) responses DataFrame.

Returns (float) the R^2 value.

Example:

```
        from pyfora.algorithms import GradientBoostedRegressorBuilder

        builder = GradientBoostedRegressorBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})

        model = builder.fit(x, y)

        # compute the score of the fit model:
        model.score(x, y)
```

class pyfora.algorithms.regressionTrees.GradientBoostedRegressorBuilder.**IterativeFitter**(*model*, *predictions*)

A sort of iterator class which is capable of fitting subsequent boosting models.

model
the current regression model.

predictions
the current predictions of the regression model (with respect to the training set implicit in `model`).

Note: This class is typically not instantiated directy. Instead these classes are returned from *iterativeFitter()*.

---

next()

Fit one boosting stage, returning a new *IterativeFitter* object that holds the next regression model and predictions.

Examples:

```
        from pyfora.algorithms import GradientBoostedRegressorBuilder

        builder = GradientBoostedRegressorBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})
```

```
        fitter = builder.iterativeFitter(x, y)

        # compute scores vs number of boosts
        numBoosts = 5
        scores = []
        for ix in xrange(numBoosts):
            fitter = fitter.next()
            scores = scores + [fitter.model.score(x, y)]
```

class pyfora.algorithms.regressionTrees.GradientBoostedClassifierBuilder.**GradientBoostedClass**

A class which builds (or "fits") gradient boosted (regression) trees to form classification models.

Parameters

- **maxDepth** (*int*) – The max depth allowed of each constituent regression tree.
- **nBoosts** (*int*) – The number of boosting iterations used.
- **learningRate** (*float*) – The learning rate of the model, used for regularization. Each successive tree from boosting stages are added with multiplier `learningRate`.
- **minSamplesSplit** (*int*) – The minimum number of samples required to split a regression tree node.
- **numBuckets** (*int*) – The number of buckets used in the estimation of optimal column splits for building regression trees.
- **loss** – the loss used when forming gradients. Defaults to `l2`, for least-squares loss. The only other allowed value currently is `lad`, for "least absolute deviation" (aka `l1-loss`).

Note: Only `nClasses = 2` cases are currently supported.

---

fit (*X*, *y*)

Fit predictors `X` to responses `y`.

Parameters

- **X** (`pandas.DataFrame`) – predictors.
- **y** (`pandas.DataFrame`) – responses.

Returns a *BinaryClassificationModel*

Examples:

```
        from pyfora.algorithms import GradientBoostedClassifierBuilder

        builder = GradientBoostedClassifierBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
```

```
        y = pandas.DataFrame({'y': [0,1,1]})

        model = builder.fit(x, y)
```

iterativeFitter(*X*, *y*)

Create an *IterativeFitter* instance which can iteratively fit boosting models.

Parameters

- **X** (DataFrame) – predictors.
- **y** (DataFrame) – responses.

Examples:

```
        from pyfora.algorithms import GradientBoostedClassifierBuilder

        builder = GradientBoostedClassifierBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})

        fitter = builder.iterativeFitter(x, y)

        numBoosts = 5
        for ix in xrange(numBoosts):
            fitter = fitter.next()
```

class pyfora.algorithms.regressionTrees.BinaryClassificationModel.**BinaryClassificationModel**(*a*
X
c
X
n
s
y
l
b
M
e
E
l
i

A class representing a gradient-boosted binary classification tree model fit to data.

Note: These classes are not normally instantiated directly. Instead, they are typically returned by *GradientBoostedClassifierBuilder* instances.

---

deviance(*x*, *yTrue*)

Compute the binomial deviance (average negative log-likihood) of the instances in predictors X with responses y.

Parameters

- **x** – the predictor DataFrame.
- **yTrue** – the (true) responses DataFrame.

Examples:

```
builder = GradientBoostedClassifierBuilder(1, 1, 1.0)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
y = pandas.DataFrame({'y': [0,1,1]})

model = builder.fit(x, y)

# compute the deviance:
model.deviance(x, y)
```

predict(*df*)

Predict the class labels of the rows of `df`.

Parameters **df** (`pandas.DataFrame`) – input DataFrame.

Returns A `pandas.Series` giving the row-wise predictions.

Examples:

```
builder = GradientBoostedClassifierBuilder(1, 1, 1.0)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
y = pandas.DataFrame({'y': [0,1,1]})

model = builder.fit(x, y)

# use the fit model to predict `x` itself:
model.predict(x)
```

predictProbability(*df*)

Return class-zero probability estimates of the rows of a DataFrame `df`.

Parameters **df** (`pandas.DataFrame`) – input DataFrame.

Returns A `pandas.Series` giving the row-wise estimated class-zero probability estimates

Examples:

```
builder = GradientBoostedClassifierBuilder(1, 1, 1.0)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
y = pandas.DataFrame({'y': [0,1,1]})

model = builder.fit(x, y)

# use the fit model to predict `x` itself:
model.predictProbability(x)
```

score(*x*, *yTrue*)

Compute the mean accuracy in predicting `x` with respect to `yTrue`.

Parameters

- **x** – the predictor DataFrame.
- **yTrue** – the (true) responses DataFrame.

Examples:

```
builder = GradientBoostedClassifierBuilder(1, 1, 1.0)
x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
y = pandas.DataFrame({'y': [0,1,1]})

model = builder.fit(x, y)
```

```
        # use the fit model to predict `x` itself:
        model.score(x, y)
```

class pyfora.algorithms.regressionTrees.GradientBoostedClassifierBuilder.**IterativeFitter**(*mode previousRegressionValues*)

A sort of iterator class which is capable of fitting subsequent boosting models.

model
the current regression model.

predictions
the current predictions of the regression model (with respect to the training set implicit in `model`).

Note: This class is typically not instantiated directy: instead these classes are returned from *iterativeFitter()*.

---

next()
Boost once and return a new *IterativeFitter*

Returns A *IterativeFitter* instance.

Example:

```
        from pyfora.algorithms import GradientBoostedClassifierBuilder

        builder = GradientBoostedClassifierBuilder(1, 1, 1.0)
        x = pandas.DataFrame({'x0': [-1,0,1], 'x1': [0,1,1]})
        y = pandas.DataFrame({'y': [0,1,1]})

        fitter = builder.iterativeFitter(x, y)

        # compute scores vs number of boosts
        numBoosts = 5
        scores = []
        for ix in xrange(numBoosts):
            fitter = fitter.next()
            scores = scores + [fitter.model.score(x, y)]
```

Data Frames

pyfora.pandas_util.**read_csv_from_string**(*data*)
Reads a string in CSV format into a DataFrame. This function is similar to `pandas.read_csv()` but it takes a string as input instead of a file.

This function is intended to be used in pyfora code that runs remotely in a pyfora cluster.

Parameters **data** (*str*) – a string of comma-separated values

Returns A `pandas.DataFrame` that holds the parsed data.

Note: This function currently assumes that all values are of type float (or floatifiable), and that the first row contains column headers. This limitation will be removed in the near future.

1.3.10 pyfora.aws

- genindex

p