More Meta

Api generation, factories, … and speed!

Dr. Scott Gorlin

Harvard University

Fall 2018

Agenda

- Midterm
- Api's and Clients
- Factories
- Optimization
- Final Project
- Readings

Midterm

HAVE NO FEAR, THE CURVE IS HERE

Api's and Clients

The API

A simple, structured way of interacting with your program

A way for microservices to communicate

For serving data via ORM…

… or configuring your program!

… or returning a model prediction!

HTTP Methods

CRUD - Create, Retrieve, Update, Delete

GET

Retrieve information

```
GET /addresses/1
```

POST

Take an action (***not idempotent***)

```
POST /addresses
```

PUT

Store or modify (eg put) data (***idempotent***)

```
PUT /addresses/1
```

DELETE

Remove a resource

```
DELETE /addresses/1
```

API Views

DRF makes it easy to implement related methods...

```
class SnippetList(APIView):

    def get(self, request, format=None):
        snippets = Snippet.objects.all()
        serializer = SnippetSerializer(snippets, many=True)
        return Response(serializer.data)
```

```
    def post(self, request, format=None):
        serializer = SnippetSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(
                serializer.data,
                status=status.HTTP_201_CREATED)
        return Response(
            serializer.errors,
            status=status.HTTP_400_BAD_REQUEST)
```

Api Views

… whether for ORM or other data!

```
class MLModelView(APIView):
    def __init__(self, pickle_file):
        self.model = open(pickle_file).read()

    def get(self, request, format=None):
        ser = SklearnSerializer(self.model)
        return Response(ser.data)
```

```
    def post(self, request, format=None):
        ser = PredictionSerializer(
            self.model.predict(request.data)
        )
        return Response(
            ser.data
        )
```

Requests

A best-of-breed generic requests library

Use whenever you want to write you own python client against any API

... but should you??

```
>>> r = requests.get(
    'https://api.github.com/user',
    auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
```

```
>>> r.json()
{
    'private_gists': 419,
    'total_private_repos': 77,
    ...
}
```

DRF CoreAPI

By Default, Django Rest Framework exposes a ‘CoreAPI’ specification

Schema generation is what makes this ***self documenting***

```
from rest_framework.documentation import
include_docs_urls

urlpatterns = [
    ...
    url(r'^docs/', include_docs_urls(title='My API
title'))
]
```

Endpoints:

```
/docs/
/docs/schema.js  # Programatic Specification
```

DRF CoreAPI

Specification is sufficient to generate an automatic client

CLI

```
$ pip install coreapi-cli
$ coreapi get http://api.example.org/
$ coreapi action users list
[
    {
        "url": "http://127.0.0.1:8000/users/2/",
        "id": 2,
        "username": "aziz",
        "snippets": []
    },
    ...
]
```

Python

```
$ pip install coreapi
```

```
import coreapi
client = coreapi.Client()
schema = client.get('https://api.example.org/')
users = client.action(schema, ['users', 'list'])
```

Works well enough… but unclear value add vs `requests`

Actions are all data, so it doesn't help much w/ development!

DRF -> Swagger

DRF-YASG

```
# urls.py
from drf_yasg.views import get_schema_view
from drf_yasg import openapi


schema_view = get_schema_view(
   openapi.Info(
      title="Snippets API"
   ),
)
```

```
urlpatterns = [
   url(r'^swagger.json$', schema_view.without_ui(),
       name='schema-json'),
   ...
]
```

Django Rest Swagger

```
from django.conf.urls import url
from rest_framework_swagger.views import
get_swagger_view


schema_view = get_swagger_view()

urlpatterns = [
    url(r'^$', schema_view)
]
```

DRF -> Swagger

Future DRF

> *We're planning to iteratively working towards OpenAPI becoming the standard schema representation … coreapi will gradually become removed, and we'll instead generate the schema directly.*
>
> — *DRF What's Next*

Swagger Clients

Makefile:

```
# Generate client, put in repo
swagger-codegen generate -l python -i swagger.json -o cortex_swagger
```

Swagger Clients

```
pip install cortex_client
```

```
from cortex_client import apis as ctx_apis, models as ctx_models
from cortex_client.api_client import ApiClient

client = ApiClient(host='cortex.lmig.com')
api = ctx_apis.EMRCreateAPI(client)
```

```
cluster = ctx_models.EmrCreateClusterRequest(
    name='test_cluster',
    count=5,
    ...
)
response = api.create_cluster(cluster)
print("Created {}".format(response.job_flow_id))
```

Swagger Clients

cortex_client.apis.emr_create_api

```
class EMRCreateApi(object):
    """This class is auto generated"""

    def create_cluster(self, auth, model, **kw):
        """createCluster

        :param str auth: Cortex auth (required)
        :param EmrCreateClusterRequest model: model
            (required)

        :return: RunJobFlowResult
        """
        ...
```

cortex_client.models.emr_create_cluster_request

```
class EmrCreateClusterRequest(object):

    def __init__(self, name=None, count=None, ...):
        ...

    @property
    def name(self):
        """Friendly name of cluster for display"""
        ...

    @property
    def count(self):
        """Total number of nodes (incl master)"""
        ...
```

Factories

Constructors

A class provides a means of ***constructing*** an object:

```
class Student:
    def __init__(self, name):
        ...

    def take_exam(self):
        ...
```

Constructors

… but that's only a default

```
class Student:
    @classmethod
    def from_records(cls, names):
        return [
            cls(name)
            for name in names
        ]
```

Creation Design Patterns

Abstract Factory

An interface for creating families of related objects without specifying their concrete classes

Builder

Separates the construction of a complex object from its representation

Factory Method

Defines an interface for creating an object, but let subclasses decide which class to instantiate

Prototype

Uses a prototype instance, which is cloned to produce new objects

Simple Factories

Eg, `sklearn.datasets`

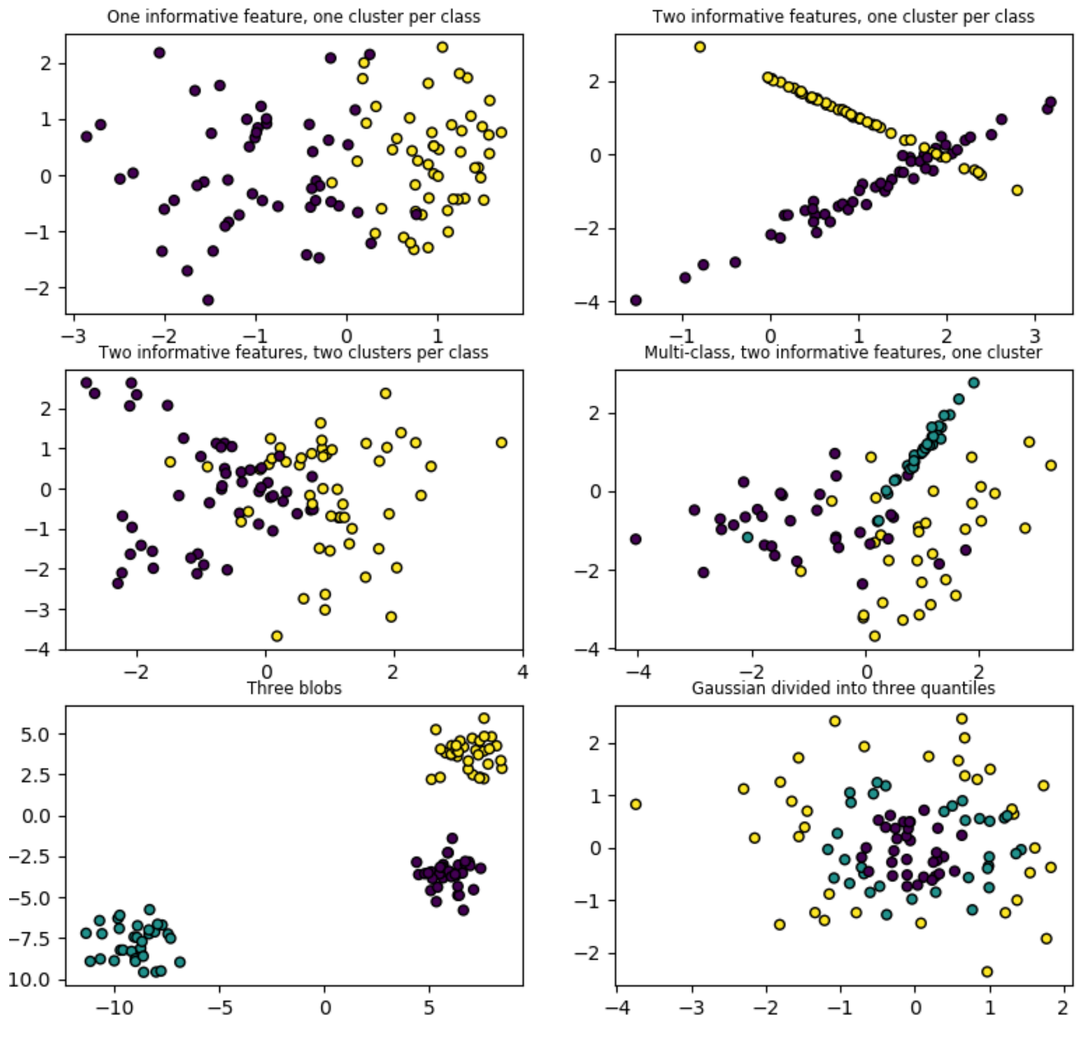
Functions designed to construct data blobs for testing classifiers

```
# sklearn.datasets

def make_classification(n, p, nclasses, ...):
    """Generate a random classification problem.

    :returns: x[n, p], y[n]
    """

def test_clf():
    x, y = make_classification(100, 2)
    clf = MyClassifier()
    clf.train(x, y)
    ...
```

Simple Factories

Eg, `sklearn.datasets`

Functions designed to construct data blobs for testing classifiers



Prototype Factories

Eg luigi's `Task.clone`

```
class Task:
    def clone(self, other_cls, **kwargs):
        new_k = {}
        for param_name, param_class in other_cls.get_params():
            if param_name in kwargs:
                new_k[param_name] = kwargs[param_name]
            elif hasattr(self, param_name):
                new_k[param_name] = getattr(self, param_name)

        return other_cls(**new_k)
```

Prototype Factories

... or our HistoryRequirement

```
class HistoryRequirement(Requirement):
    def __call__(self, task):
        d = task.date
        return [
            task.clone(self.other_task, date=d-timedelta(days-i-1))
            for i in range(self.ndays)
        ]
```

Factory Methods

Best achieved with `@classmethod`

Provide a new, common way to create instances of any subclass

```
class WordEmbedding:
    @classmethod
    def from_files(cls, word_file, vec_file):
        """Instantiate an embedding from files"""

        # note cls vs WordEmbedding
        return cls(
            load_vocab(word_file),
            np.load(vec_file)
        )
```

Builders

… are classes designed to iteratively build another object

```
class Car:
    ...

class CarBuilder(Builder):
    def __init__(self):
        self.car = Car()

    def set_wheels(self, value):
        self.car.wheels = value
        return self

    def get_result(self):
        return self.car
```

```
>>> CarBuilder().set_wheels('dubs').get_result()
Car()
```

Builders

… are classes designed to iteratively build another object

Not a common DS paradigm

IMHO, except when you consider internal objects like DAG's or eg the pandas interface

```
>>> ddf = dask.dataframe.read_parquet('data/grad')
>>> ddf.dask
{
    ('read-parquet-73a212', 0): (
        _read_parquet_row_group,
        'data/hashed/grad=False/part.0.parquet',
    )
    ...
}
```

```
>>> ddf.groupby('grad').sum().dask
{
    ('dataframe-groupby-sum-agg-9939d0', 0): ...
    ...
}
```

Abstract Factories

Class designed to generate other classes

Usually subclass or parameterize to define the class to create

```
class DatasetGenerator:
    def __init__(self, **make_kwargs):
        self.make_kwargs = make_kwargs

    def __call__(self):
        return make_classification(
            **self.make_kwargs)

>>> factory = DatasetGenerator(nclasses=3)
>>> factory()
([[...]], [...])
```

Factory Boy

Easy factories for ORM's (and any class)

Claims to be for testing, but is useful in general as well!

```
from factory import Factory

class UserFactory(Factory):
    class Meta:
        model = User

    firstname = "John"
    lastname = "Doe"

>>> john = UserFactory()
<User: John Doe>
```

Factory Boy

Note that syntax!

Factory Boy made a ***class*** constructor return an instance of a ***different*** class!!

```
class FactoryMetaClass(type):
    # __call__ as a classmethod!
    def __call__(cls, **kwargs):
        # Normally, cls() would create an
        # instance of the class

        if cls._meta.strategy == BUILD_STRATEGY:
            return cls.build(**kwargs)
        ...
```

```
# cf (not how they implemented it!):
>>> factory = UserFactory()
# instance.__call__()
>>> john = factory()
<User: John Doe>
```

Factory Boy

Note that syntax!

Arguably, they should have separated defaults and constructors

```
# Not how they did it!

# factory definition: schema/model etc
class UserFactory(Factory):
    class Meta:
        model = User
```

```
# Create defaults with an instance, not subclass
>>> factory = UserFactory(firstname='John')
>>> factory()  # Fake/random data
<User: John Kennedy>
>>> factory(lastname='Doe') # Overrides
<User: John Doe>
>>> factory(lastname='Connor')
<User: John Connor>
```

Factory Boy

Designed for Django, but promoted/generalized to work with any ORM or object type

```python
class OrderTests(TestCase):
    def test_orders(self):
        order = OrderFactory(
            amount=200,
            status='PAID',
            # Automatically creates FK's
            customer__is_vip=True,
            address__country='AU',
        )
        # Run the tests here
```

Factory Boy

Generic

```
class UserFactory(Factory):
    class Meta:
        model = objects.User

    username = Sequence(lambda n: 'john%s' % n)
    date_joined = LazyFunction(datetime.now)
```

Django

```
class UserFactory(DjangoModelFactory):
    class Meta:
        model = 'myapp.User'
        django_get_or_create = ('username',)

    username = 'john'
```

Alchemy

```
class UserFactory(SQLAlchemyModelFactory):
    class Meta:
        model = User
        sqlalchemy_session = session

    id = Sequence(lambda n: n)
    name = Sequence(lambda n: u'User %d' % n)
```

Faker

Factory Boy uses Faker to help stub out fake data for testing

```
from faker import Faker
fake = Faker()

>>> fake.name()
'Lucy Cechtelar'

>>> fake.address()
'426 Jordy Lodge Cartwrightshire, SC 88120-6700'
```

The Point

Classes provide reusable pairings of data and functions

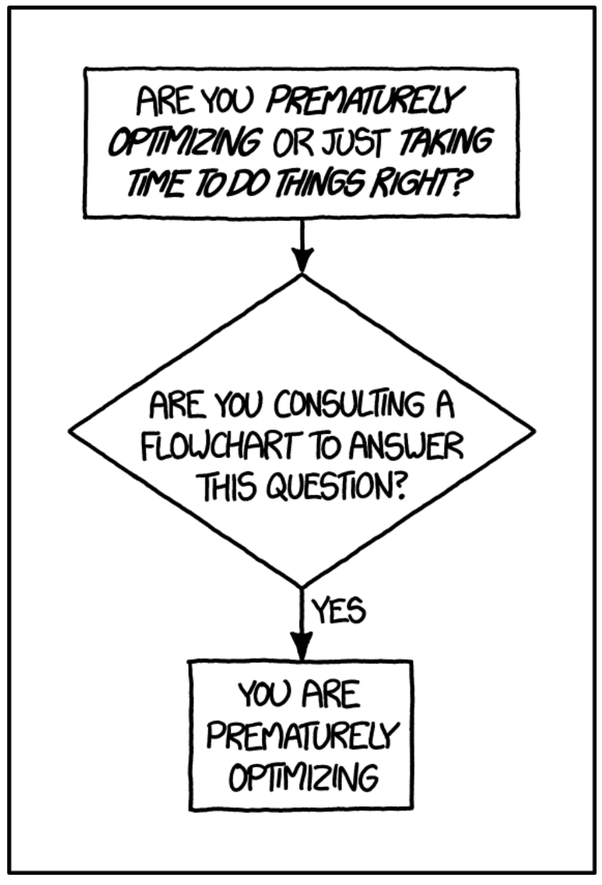
Factories provide reusable constructors for classes that can encapsulate various and more sophisticated creation strategies

If `MyClass()` isn't enough to create a good default instance for you, consider a factory

Optimization

Root of all Evil

Never, ever optimize your code until you know enough to break my rules



XKCD 1691

Root of all Evil

Optimization should be restricted to vectorization and appropriate use of libraries

```python
def dist(a, b):
    # M(1)
    out = 0
    for i in range(a.shape):
        out += (a[i] - b[i]) ** 2
    return out ** 0.5
```

```python
def dist(a, b):
    # M(|a|)
    # Extra memory overhead likely worth it
    # for numpy vectorization and clarity
    return (
        ((a - b) ** 2).sum()
    ) ** 0.5
```

Tale of Two Compilers

If you ***absolutely*** need the speed…

Raw C/Fortran

Only for existing libraries

Can write python bridge

Cython

Python-like code

Complies via C on install

Great interface to other C libs (incl numpy)

Numba

Pure python code

JIT compiled to C/GPU

Unclear stability/adoption

Cython

> *Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex). It makes writing C extensions for Python as easy as Python itself.*
>
> — *Cython*

Cython

```
def dist(a, b):
    out = 0
    for i in range(len(a)):
        out += (a[i]-b[i])**2
    return out**0.5

a = np.random.randn(10000)
b = np.random.randn(10000)
```

```
%timeit dist(a, b)
# 5.98 ms ± 587 µs per loop
```

Vanilla Cython

```
%%cython
def dist_cython(a, b):
    out = 0
    for i in range(len(a)):
        out += (a[i]-b[i])**2
    return out**0.5
```

```
%timeit dist_cython(a, b)
# 4.82 ms ± 629 µs (25%)
```

Static Cython

```
%%cython
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def dist_static(double[:] a,
                double[:] b):
    cdef:
        double out = 0.
        int i

    for i in range(len(a)):
        out += (a[i]-b[i])**2
    return out**0.5
```

```
%timeit dist_static(a, b)
# 14.6 µs ± 972 ns (410x!!)
```

Cython

Static Cython

```
%%cython
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def dist_static(double[:] a,
                double[:] b):
    cdef:
        double out = 0.
        int i

    for i in range(len(a)):
        out += (a[i]-b[i])**2
    return out**0.5
```

```
%timeit dist_static(a, b)
# 14.6 µs ± 972 ns (410x!!)
```

Flex vs Speed

- `double[:] a`: only 1D floating point arrays
  - Can even `double[::1]` for contiguous memory
- `boundscheck`: segfault instead of `IndexError`
- `wraparound`: no `a[-1]`
- `cdef double`: specify c types

Cython

Static Cython

```
%%cython
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def dist_static(double[:] a,
                double[:] b):
    cdef:
        double out = 0.
        int i

    for i in range(len(a)):
        out += (a[i]-b[i])**2
    return out**0.5
```

```
%timeit dist_static(a, b)
# 14.6 µs ± 972 ns (410x!!)
```

Numpy

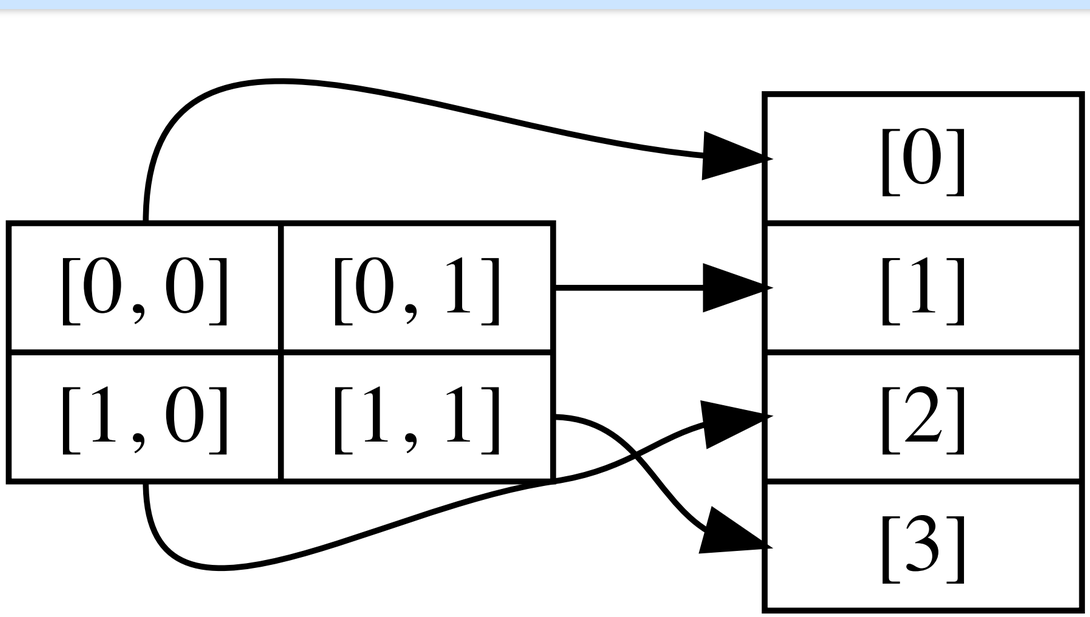
```
def dist_np(a, b):
    return ((a-b)**2).sum() ** 0.5
# 20.8 µs ± 1.84 µs (290x)
```

Cython is 40% faster and M(1) vs numpy M(N), but at great cost!

Ensure your problem needs this kind of optimization before trying!

Memory Views

C-contiguous data: values next to each other on the ***last*** dimension are adjacent in memory




```
>>> a = np.arange(4).reshape(2, 2)
array([[0, 1],
       [2, 3]])

>>> a.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False

>>> a.strides
(16, 8) # Bytes to advance each dim
```

Contiguous iterations are faster!

```
m = np.random.randn(500, 500)
%timeit m.sum(axis=0)
# 93.9 µs ± 346 ns
%timeit m.sum(axis=1)
# 76 µs ± 689 ns
```

Memory Views

Yes, that means R (and pandas) are faster for columnar analytics!

```
>>> df = DataFrame(
    {'a':range(5),
     'b':range(0, 10, 2)})

>>> df.values
array([[0, 0],
       [1, 2],
       [2, 4],
       [3, 6],
       [4, 8]])
```

```
>>> df.values.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True

>>> df['a'].values.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : True
```

Memory Views

Much of numpy's speed is because arrays are pointers to the same data, with tweaks on their strides!

Numpy will only copy the data if it needs to - many operations result in *new views*

```
>>> a.flags.owndata
True
>>> a.T.flags.owndata
False
```

```
# Transpose switches the stride order
# but views the same data!
>>> a.strides
(8, 16)
>>> a.T.strides
(16, 8)
```

```
# Reversing a dimension moves the
# head pointer to the end, and
# negates the strides!
>>> a[:, ::-1].strides
(16, -8)
```

Memory Views

Numpy allows for various (non) contiguous views…

```
>>> np.arange(4).reshape(2,2).T.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : False
```

```
>>> np.arange(0, 32, 2).flags
C_CONTIGUOUS : True
F_CONTIGUOUS : True
>>> np.arange(32)[::2].flags
C_CONTIGUOUS : False
F_CONTIGUOUS : False
```

```
>>> np.arange(5)[::-1].strides
(-8,)
```

Cython can force contiguity

```
def flexifunc(double[:] array):
    """Any 1-d array interface"""
    ...

def superfast(double[::1] array):
    """Forces c-contiguous array"""
    ...
```

… just be ***absolutely*** sure you need this! You probably don't!

Cython Standard

Cython is the de-facto standard for OSS libraries

```
# sklearn/linear_model/stochastic_gradient.py
from .sgd_fast import plain_sgd

class SGD:
    def fit_binary(self, X, y):
        return plain_sgd(
            self.coef_, X, y)
```

```
# sklearn/linear_model/sgd_fast.pyx
def plain_sgd(...):
    with nogil:
        for epoch in range(max_iter):
            for i in range(n_samples):
                dataset.next(
                    &x_data_ptr, &x_ind_ptr, &xnnz,
                    &y, &sample_weight)
                p = w.dot(
                    x_data_ptr, x_ind_ptr, xnnz
                    ) + intercept
```

Cython Standard

Cython is the de-facto standard for OSS libraries

… because Cython is only needed to generate C code to build wheel; it is not necessary for installation!

But it doesn't always get first class support.

Sklearn doesn't uniformly include cython .pxd (header) files, which are necessary for subclassing/overriding/extending the cython implementations

Cython Standard

scikit-learn / scikit-learn

Watch 2,209 | Unstar 31,685 | Fork 15,638

Code | Issues 1,143 | Pull requests 662 | Projects 5 | Wiki | Insights

[MRG] Fix #12363 Included criterion.pxd #12381

Merged rth merged 4 commits into `scikit-learn:master` from `RoopamSharma:Included_criterion.pxd` 14 days ago

Conversation 6 | Commits 4 | Checks 0 | Files changed 1

Changes from all commits | Jump to... | +4 −0 | Diff settings | Review changes

4 sklearn/tree/setup.py — View

```
@@ -31,6 +31,10 @@ def configuration(parent_package="", top_path=None):
                           extra_compile_args=["-O3"])

     config.add_subpackage("tests")
+    config.add_data_files("_criterion.pxd")
+    config.add_data_files("_splitter.pxd")
+    config.add_data_files("_tree.pxd")
+    config.add_data_files("_utils.pxd")

     return config
```

GIL and Parallelism

The global interpreter lock, or GIL, … protects access to Python objects, preventing multiple threads from executing Python bytecodes at once.

— *Python Wiki*

Multi-Threading in python has no performance benefit …

… unless you release the GIL!

GIL and Parallelism

Cython allows for explicit release of the GIL and true thread parallelism with `prange` via OpenMP

```
from cython.parallel import prange

def func(double[:] x, double alpha):
    cdef int i
    with nogil:
        for i in prange(x.shape[0]):
            x[i] = alpha * x[i]
```

Numba

JIT

Numba is a Just In Time compiler

It takes python and compiles it to native code via LLVM ***at runtime***

```
from numba import jit
dist_nb = jit(dist)
# 12.7 µs ± 68.1 ns (470x!!!)
```

Holy smokes Batman, instant speed with no code change!

Small runtime latency, but I don't need to compile in setup.py!

Numba works well in conda, but historically not with pip due to LLVM requirement. The docs claim this is resolved.

Numba

But wait! There's more!!

Simplified Threading

```
@jit(nopython=True,
parallel=True)
def simulator(out):
    # iterate loop in parallel
    for i in prange(len(out)):
        out[i] = run_sim()
```

Cython does this too

SIMD Vectorization

```
LBB0_8:
    vmovups (%rax,%rdx,4), %ymm0
    vmovups (%rcx,%rdx,4), %ymm1
    vsubps  %ymm1, %ymm0, %ymm2
    vaddps  %ymm2, %ymm2, %ymm2
```

GPU Acceleration

Numba

Instant numpy ufuncs!

```
from numba import vectorize, float64

@vectorize([float64(float64, float64)])
def f(x, y):
    return x + y
```

```
# Broadcasting
>>> a = np.linspace(0, 1, 3)
array([0. , 0.5, 1. ])

>>> f(a.reshape(-1, 1), a.reshape(1, -1))
array([[0. , 0.5, 1. ],
       [0.5, 1. , 1.5],
       [1. , 1.5, 2. ]])
```

```
# ufunc specials
>>> f.reduce(a)
1.5
```

Numba

Numba is promising and much lower barrier to entry. You can use pure python and get highly optimized code.

It is developed by Continuum/Anaconda

Adoption is not high, but is increasing. It may not be stable. Be vigilant.

Final Project

12/12 Science Fair

12/19 is the final exam

12/12 is the last content class

12/12 will be an interactive science fair. You will present and defend your final project to your peers

Be prepared to present to small groups from your laptop (or a poster…) for 5-10 minutes. We will do several rounds to ensure everyone can discuss every project.

Thanksgiving

No class next week. Enjoy your break!

Readings

- Understanding REST
- Creational Patterns
- Factory Method Pattern
- (old but good) Numba vs Cython
- Cython for NumPy Users
- How Numba and Cython speed up Python code
- Discussion on whether cython code is ‘private’: SKLEARN-2057
- The Case for Numba