Python Array API Standard

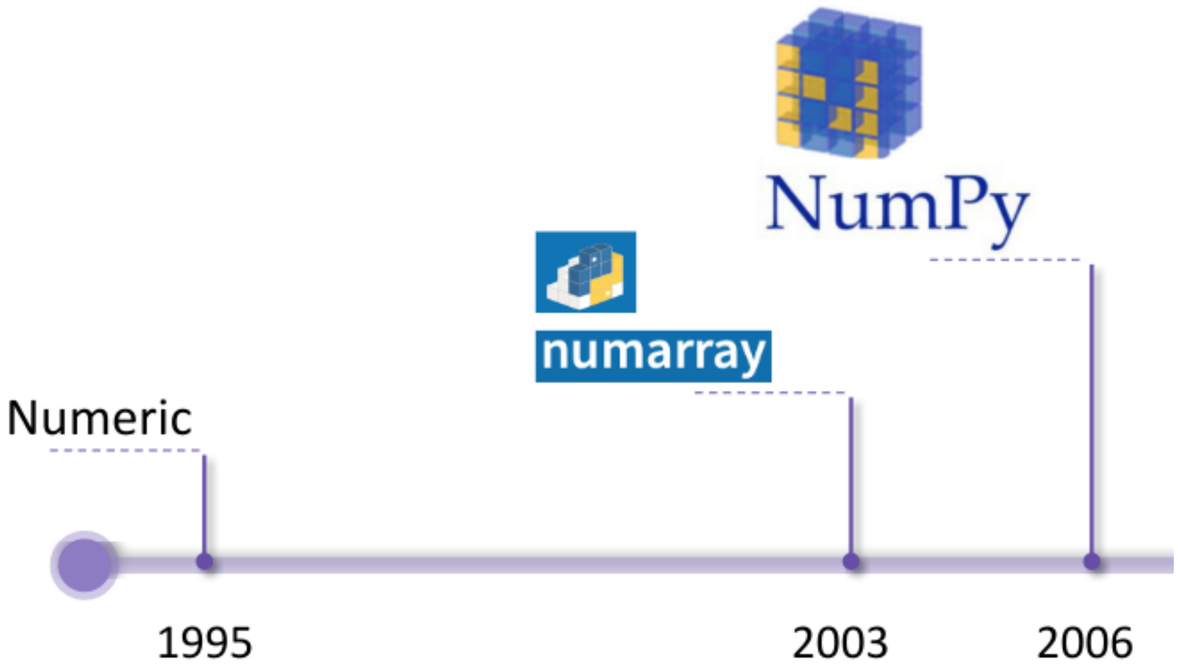
Toward Array Interoperability in the Scientific Python Ecosystem

Aaron Meurer

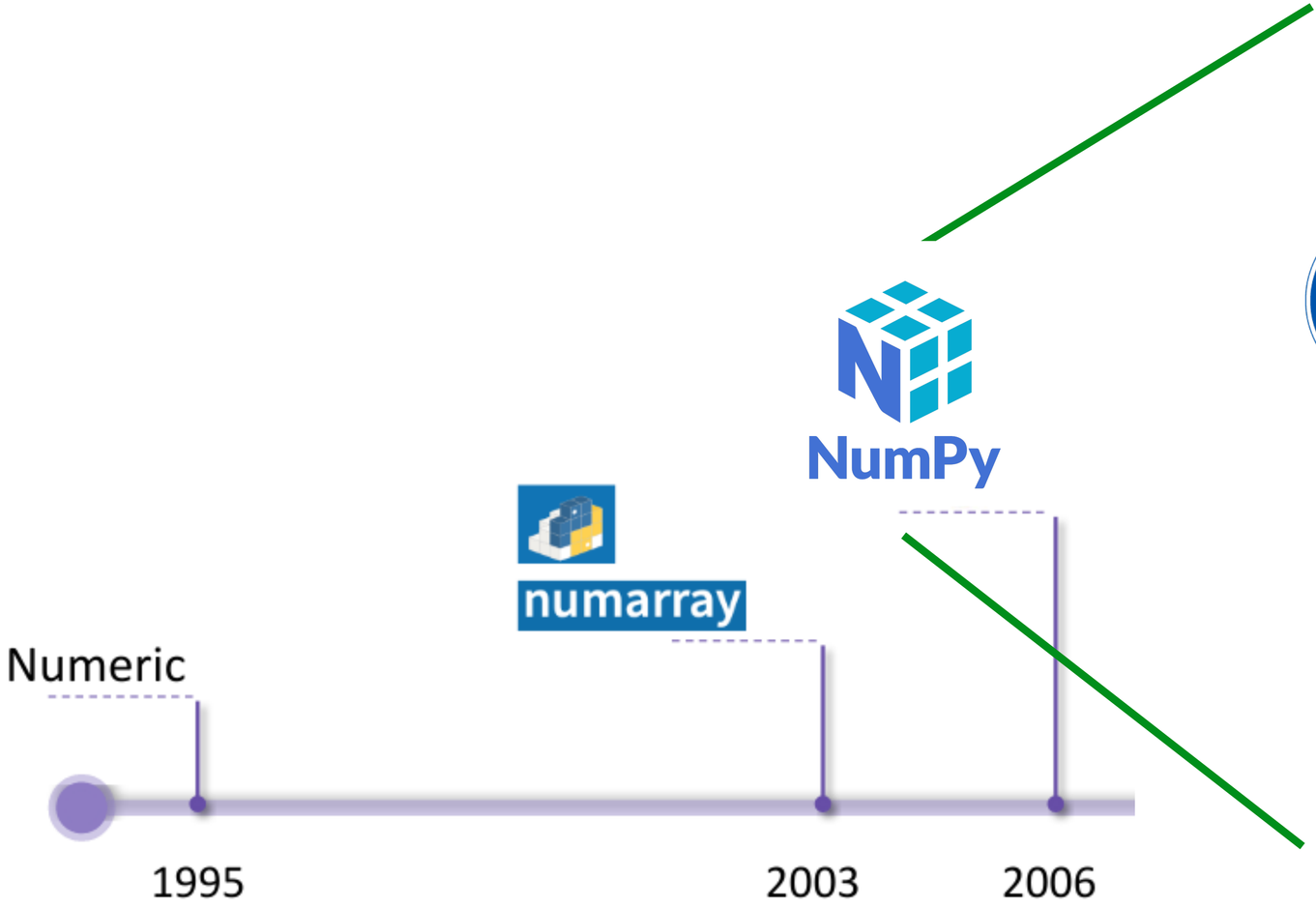
July 14, 2023

SciPy 2023, Austin, TX

In the beginning there was NumPy...



In the beginning there was NumPy...



matplotlib

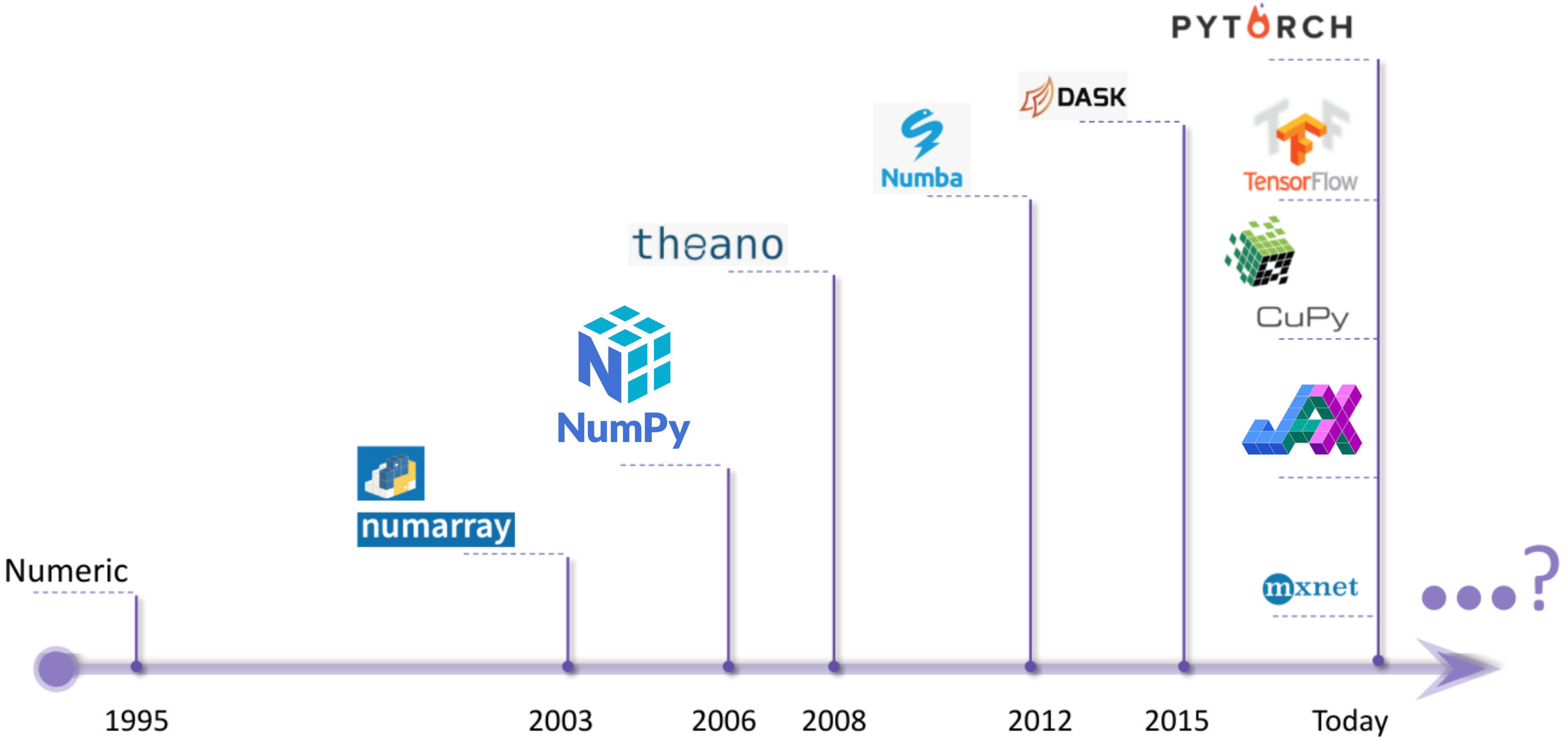
SciPy

scikit
learn

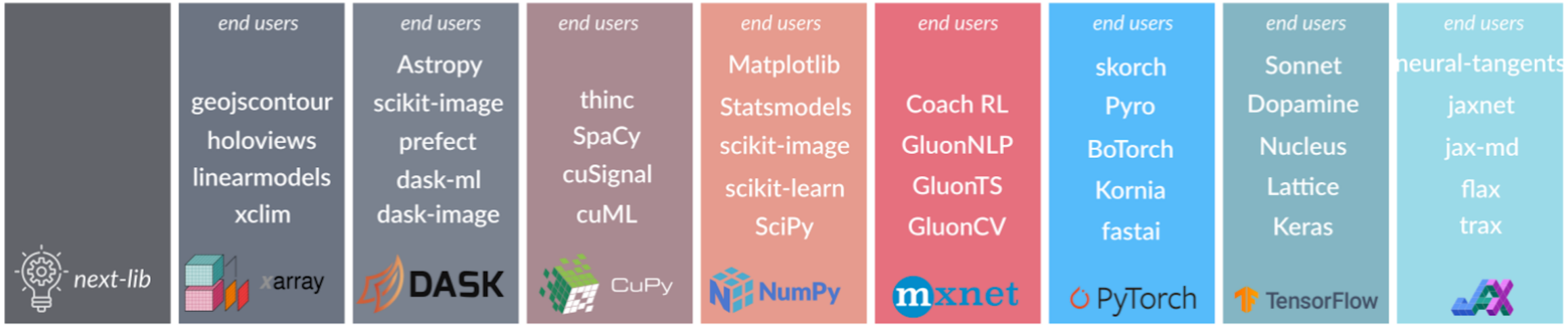
scikit-image
image processing in python

statsmodels

Today, there are many array libraries

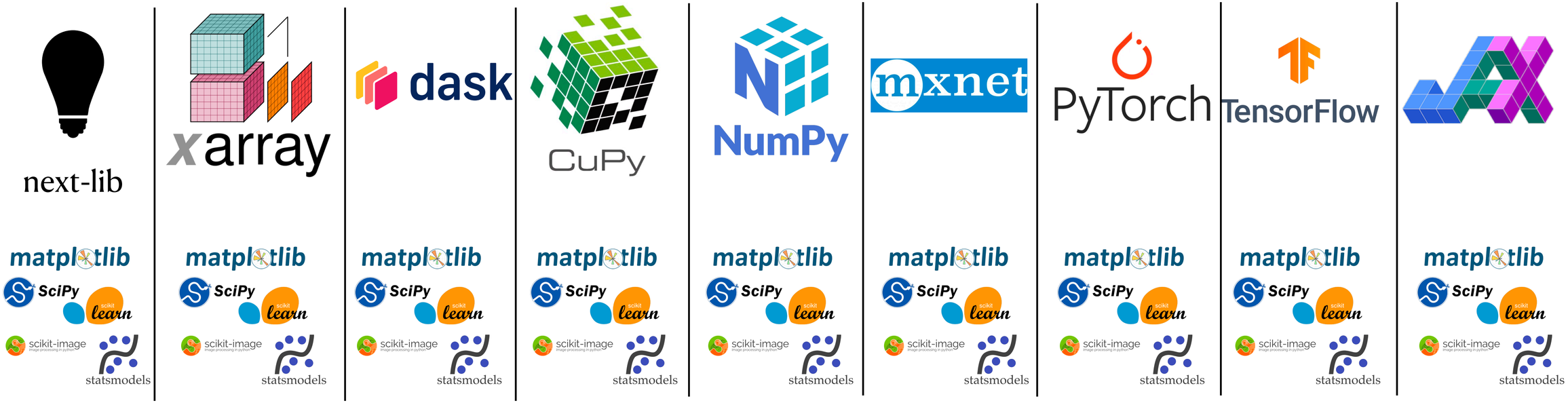


Each library has its own ecosystem



Tools written against (e.g.) NumPy cannot work with other libraries

What we'd like to see



Example: SciPy + CuPy = FAIL


```
>>> import scipy.signal
>>> import cupy
>>> x = cupy.asarray(...)
>>> scipy.signal.welch(x)
Traceback (most recent call last):
...
TypeError: Implicit conversion to a NumPy array is
not allowed. Please use `.get()` to construct a
NumPy array explicitly.
```

The problem

- SciPy (and scikit-learn, scikit-image, statsmodels, ...) are written against NumPy
- They use `import numpy as np`
- The code implicitly assumes NumPy functions and semantics.
  - CuPy closely matches the NumPy API, but other libraries like PyTorch do not.
  - Even with CuPy, code needs to avoid GPU antipatterns to be performant (e.g. unnecessary device transfers).

Potential Solutions

Potential Solutions

“Merge everything into NumPy”

Potential Solutions

“Merge everything into NumPy”

- ✗ Each library has its own set of strengths and APIs (targeted hardware, performance, distributed computation, etc.).
- ✗ This is way too much scope for NumPy.
- ✗ Sometimes these design decisions contradict one another.

Potential Solutions

“Make every library match the NumPy API” (CuPy approach)

Potential Solutions

“Make every library match the NumPy API” (CuPy approach)

- ✗ NumPy was designed around eager CPU computation.
- ✗ Many NumPy semantics are inappropriate for other use-cases (GPU, distributed, lazy evaluation, ...).
- ✗ Some NumPy design decisions are simply bad and shouldn’t be replicated.

Potential Solutions

“Use NumPy `__array_function__` dispatching”

Potential Solutions

"Use NumPy `__array_function__` dispatching"

- ✗ Not everything is implemented in NumPy (e.g., specialized deep learning functions).
- ✗ There's more to an API than just function signatures (indexing, operators, methods, broadcasting, type promotion, ...).
- ✗ Users expect `np.func(x)` to return a NumPy array.

Potential Solutions

“Implement a backend switching system”

Potential Solutions

“Implement a backend switching system”

- ✗ Maybe feasible, but would be a lot of work and maintenance.
- ✗ Array-consuming code would still need to special case different libraries for performance or other reasons.

Better Solution: API Standardization

Better Solution: API Standardization

“Define a common subset API for array libraries and standardize its semantics”

Better Solution: API Standardization

“Define a common subset API for array libraries and standardize its semantics”

- ✓ Won’t require any new runtime dependencies.
- ✓ Won’t require array libraries to know about each other.
- ✓ NumPy is not special. It's just another array library.

Better Solution: API Standardization

“Define a common subset API for array libraries and standardize its semantics”

- ✓ Can be based on what APIs are already commonly implemented.
- ✓ Can avoid/make optional APIs and semantics that break important use-cases (multi-device, lazy evaluation, ...).

Data APIs Consortium

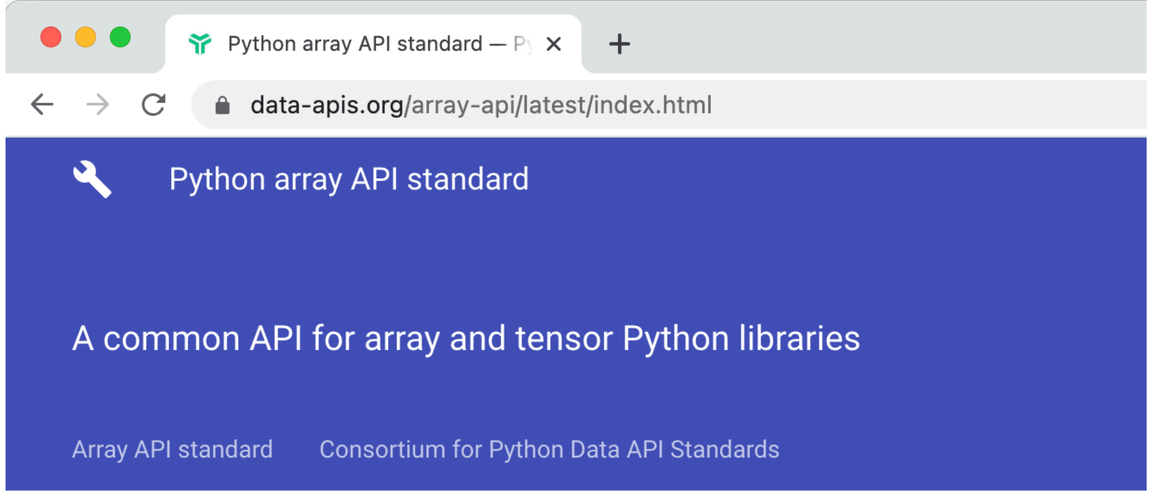
data-apis.org

- Founded in May 2020.
- A group of stakeholders from various array and dataframe libraries, industry, array-consuming libraries, and end users met regularly to discuss a standardized array API.

Array API Standard

https://data-apis.org/array-api/

- Defines a set of functions and semantics that any standards complaint array library should implement.
- No dependencies or reliance on any specific array library. It's strictly a specification.



Python array API standard

Contents

Array API Standard

Functions and Methods

- Includes ~200 functions and array methods.
- Most functions are based on already-existing APIs from common array libraries like NumPy, CuPy, PyTorch, JAX, etc.

Element-wise Functions

> Array API specification for element-wise functions.

Objects in API

abs(x, /)	Calculates the absolute value for each element `x_i` of the input array `x`.
acos(x, /)	Calculates an implementation-dependent approximation of the principal value of the inverse cosine for each element `x_i` of the input array `x`.
acosh(x, /)	Calculates an implementation-dependent approximation to the inverse hyperbolic cosine for each element `x_i` of the input array `x`.
add(x1, x2, /)	Calculates the sum for each element `x1_i` of the input array `x1` with the respective element `x2_i` of the input array `x2`.
asin(x, /)	Calculates an implementation-dependent approximation of the principal value of the inverse sine for each element `x_i` of the input array `x`.
asinh(x, /)	Calculates an implementation-dependent approximation to the inverse hyperbolic sine for each element `x_i` in the input array `x`.
atan(x, /)	Calculates an implementation-dependent approximation of the principal value of the inverse tangent for each element `x_i` of the input array `x`.
atan2(x1, x2, /)	Calculates an implementation-dependent approximation of the inverse tangent of the quotient

Example: `mean()`

- The function signature uses positional-only and keyword-only arguments for maximum portability and future-proofing.
- Type annotations only specify the minimal set of require input types. No runtime type checking is required.
- SHOULD and MUST are used following RFC 2119 ("`x` **should** have a real-valued floating-point data type", "The returned array **must** have the same data type as `x`").
- The choice of standardized keyword arguments come from usage data (more on that later).
- The precision of the output is left unspecified.

mean(*x: array*, */*, ***, *axis: Optional[Union[int, Tuple[int, ...]]] = None*, *keepdims: bool = False*) → array

Calculates the arithmetic mean of the input array `x`.

Parameters

- **x** (*array*) – input array. Should have a real-valued floating-point data type.
- **axis** (*Optional[Union[int, Tuple[int, ...]]]*) – axis or axes along which arithmetic means must be computed. By default, the mean must be computed over the entire array. If a tuple of integers, arithmetic means must be computed over multiple axes. Default: `None`.
- **keepdims** (*bool*) – if `True`, the reduced axes (dimensions) must be included in the result as singleton dimensions, and, accordingly, the result must be compatible with the input array (see Broadcasting). Otherwise, if `False`, the reduced axes (dimensions) must not be included in the result. Default: `False`.

Returns

out (*array*) – if the arithmetic mean was computed over the entire array, a zero-dimensional array containing the arithmetic mean; otherwise, a non-zero-dimensional array containing the arithmetic means. The returned array must have the same data type as `x`.

> **Note**
>
> While this specification recommends that this function only accept input arrays having a real-valued floating-point data type, specification-compliant array libraries may choose to accept input arrays having an integer data type. While mixed data type promotion is implementation-defined, if the input array `x` has an integer data type, the returned array must have the default real-valued floating-point data type.

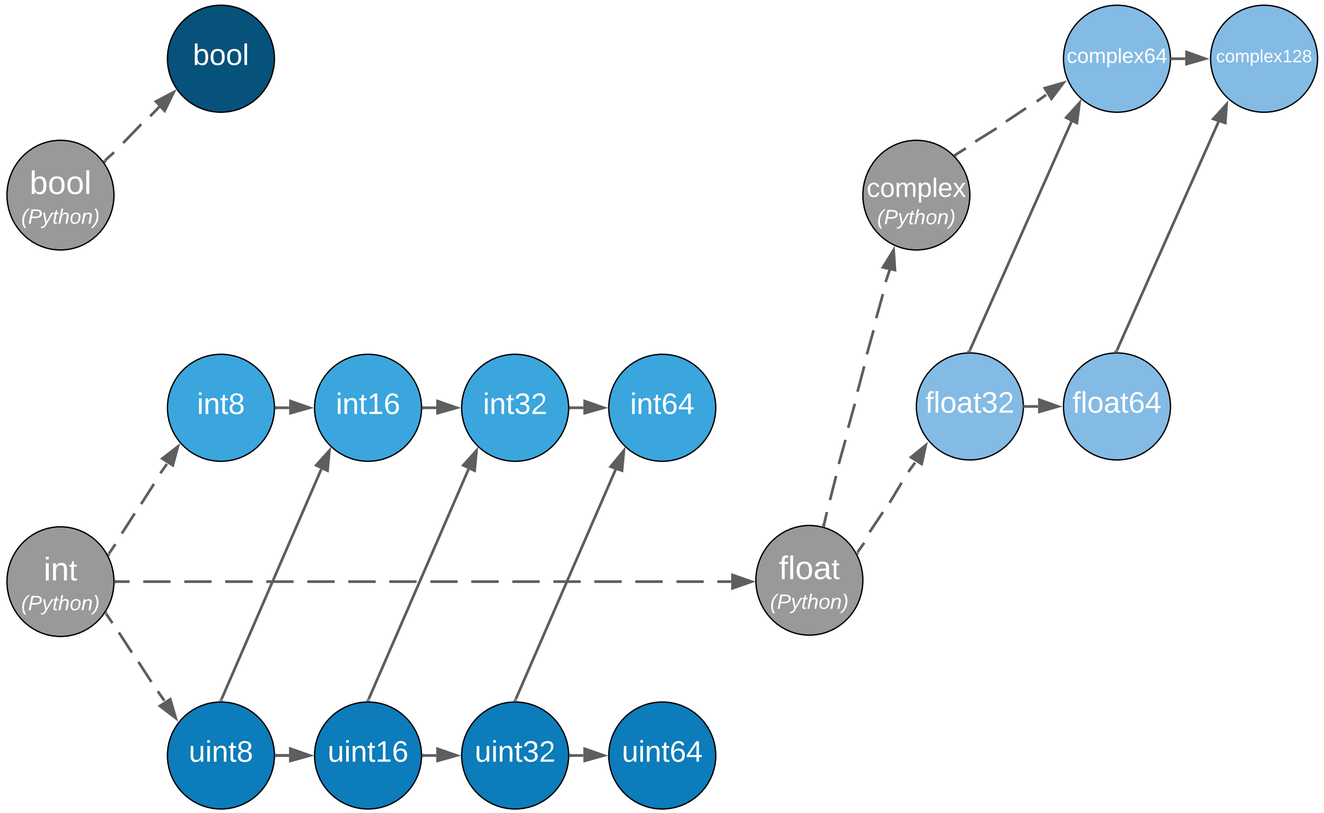
Array API Standard

Additional Semantics

- Broadcasting
- Indexing
- Type Promotion
- Interchange Protocol
- Device Support

Type Promotion

- Basic set of common numeric dtypes (including integer, real floating-point, and complex).
- Cross-kind type promotion is not required (e.g., `int8 + float32`).
- Type promotion should not depend on array values or array shape.



In-memory Interchange Protocol

- DLPack
  - Header-only dependency
  - Stable C ABI
  - Specifies zero-copy semantics
  - Support for multiple devices
  - Already implemented in most popular Python array libraries (NumPy, CuPy, PyTorch, Tensorflow, ...)

Device Support

- Three methods are implemented.
  - `.device` attribute on array objects.
  - `device=` keyword to creation functions.
  - `.to_device()` method on array objects.
- All computations should occur on the same device as the input array.
- Implicit transfers are not allowed (mixing devices should raise an exception).

Optional Extensions

- There are two optional extension namespaces
  - `linalg`
  - `fft`
- Not required as they may be difficult for some implementations.

Design Principles

Design Principles

- **Functional**
  - `xp.mean(x)` instead of `x.mean()`
- **Minimal array object**
  - Only defines basic attributes (`shape`, `dtype`, ...) & operator methods
  - No requirements about the storage of the underlying data
- **Universality**
  - Standardized functions should reflect common existing usage.
  - Only already existing APIs are specified (with a few exceptions).

Design Principles

- **Compiler support**
  - APIs should be amenable to JIT and AOT compilation.
  - APIs with data-dependent output shape are optional (like boolean masking)
  - Copy-view mutation semantics are considered a library-dependent implementation detail. Libraries may or may not support them.
- **Distributed support**
  - APIs should be amenable to distributed computation (like Dask)
- **Accelerator support**
  - Only include APIs that are implementable on GPUs and other specialized hardware

Design Principles

- **Consistency**
  - For example, use `axis` keyword argument name everywhere.
- **Extensibility**
  - The standard only specifies a *minimal* set of APIs and behaviors.
  - Libraries may choose to implement additional functions.
- **Deference**
  - Defer to other standards where possible (e.g., IEEE 754)

Methodology

Methodology

- We only want to standardize APIs that are:
  - Already implemented in most array libraries
  - Used heavily in the ecosystem
- We used a data-driven approach.

Methodology

Array API Comparison

- We compared APIs across common array libraries (including NumPy, Dask Array, CuPy, MXNet, JAX, TensorFlow, and PyTorch).
- We examined the intersection of implemented functions and their keyword arguments.
- Source code with data is available at https://github.com/data-apis/array-api-comparison

```
numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>)

cupy.mean(a, axis=None, dtype=None, out=None, keepdims=False)

dask.array.mean(a, axis=None, dtype=None, out=None, keepdims=False,
                split_every=None)

jax.numpy.mean(a, axis=None, dtype=None, out=None, keepdims=False)

mxnet.np.mean(a, axis=None, dtype=None, out=None, keepdims=False)

tf.math.reduce_mean(input_tensor, axis=None, keepdims=False,
                    name=None)

torch.mean(input, dim, keepdim=False, out=None)
```

Common Signature Subset

```
mean(a, axis=None, keepdims=False)
```

Methodology

Usage

- We runtime instrumented the test suites of common array-consuming libraries (SciPy, pandas, Matplotlib, Xarray, scikit-learn, statsmodels, scikit-image, etc.).
- Every NumPy API call was recorded, along with its input type signature.
- Each API was then ranked with the Dowdall positional voting system (a variant of the Borda count that favors APIs having high relative usage).
- Source code with data is available at https://github.com/data-apis/python-record-api

Methodology

Usage (example)

- Example output of python-record-api for `mean()`.
- This example shows that `np.mean(x)` with `x` as a `np.ndarray` is much more common than with `x` as a list of `floats`.
  - This sort of usage data helped us to understand that we should standardize array inputs to functions but not “array-like” inputs.

```
@overload
def mean(a: numpy.ndarray):
    """
    usage.dask: 21
    usage.matplotlib: 7
    usage.scipy: 26
    usage.skimage: 36
    usage.sklearn: 130
    usage.statsmodels: 45
    usage.xarray: 1
    """
```

```
@overload
def mean(a: List[float]):
    """
    usage.networkx: 6
    usage.sklearn: 3
    usage.statsmodels: 9
    """
```

Status

Specification Status

- Two released versions v2021.12 and v2022.12.
  - v2021.12 was the initial release.
  - v2022.12 added complex numbers, fast Fourier transforms extension, and a few additional functions.
- Current focus in 2023 is on adoption.

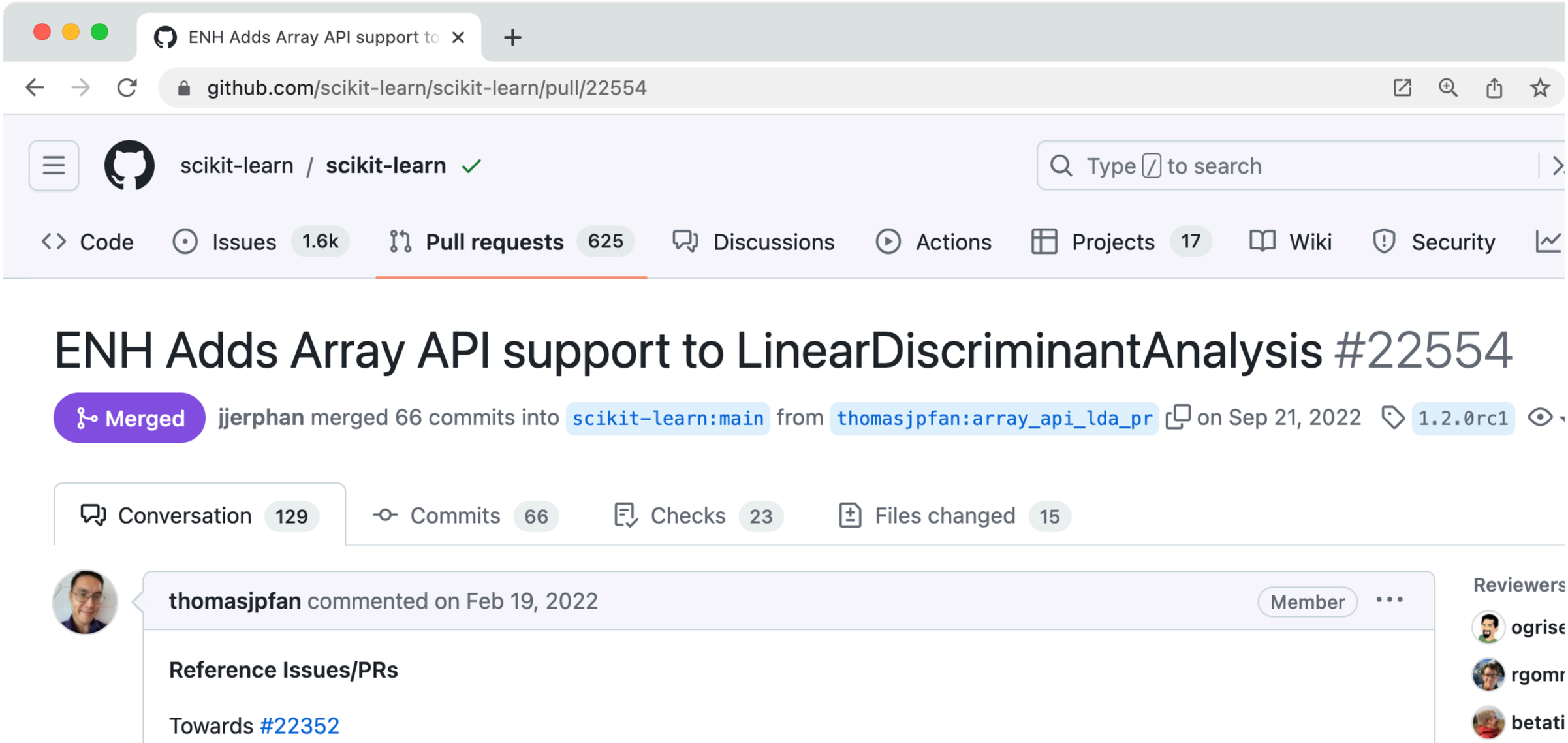
Implementation Status

- NumPy contains minimal reference implementation, `numpy.array_api` (more on that later).
- Main NumPy namespace is planned to be fully compliant for NumPy 2.0.
- CuPy will follow NumPy.
- PyTorch is mostly compliant. Has open issues for most existing differences.
- JAX implementation in progress (https://github.com/google/jax/pull/16099)
- Dask Array implementation in progress (https://github.com/dask/dask/pull/8750)

Adoption Status

- SciPy and scikit-learn are both actively moving to adopt the array API.
  - scikit-learn 1.3 will add experimental array API support to `LinearDiscriminantAnalysis`.
- If you want to chat about adopting the array API, come to my sprint tomorrow!

Adoption Example: scikit-learn



Adopted from https://github.com/scikit-learn/scikit-learn/pull/22554

The standard only specifies boolean indices as the sole index and multidimensional indexing when all axes are indexed.

`dot()` is not included in the standard. The matmul operator `@` is used instead.

The standard function is called `concat()`.

The standard uses functions on the namespace rather than array methods (`xp.std(x)` instead of `x.std()`).

Functions do not implicitly support non-array inputs. Here `asarray()` is used to convert the input to `sqrt()` to an array.

`np` is replaced with `xp` everywhere.

Numerical functions only support numerical dtypes. `sum()` on a boolean array requires an explicit conversion.

```
 Xc = []
  for idx, group in enumerate(self.classes_):
-     Xg = X[y == group, :]
-     Xc.append(Xg - self.means_[idx])
+     Xg = X[y == group]
+     Xc.append(Xg - self.means_[idx, :])

- self.xbar_ = np.dot(self.priors_, self.means_)
+ self.xbar_ = self.priors_ @ self.means_

- Xc = np.concatenate(Xc, axis=0)
+ Xc = xp.concat(Xc, axis=0)

- std = Xc.std(axis=0)
+ std = xp.std(Xc, axis=0)

  std[std == 0] = 1.0
- fac = 1.0 / (n_samples - n_classes)
+ fac = xp.asarray(1.0 / (n_samples - n_classes))

- X = np.sqrt(fac) * (Xc / std)
+ X = xp.sqrt(fac) * (Xc / std)

  U, S, Vt = svd(X, full_matrices=False)

- rank = np.sum(S > self.tol)
+ rank = xp.sum(xp.astype(S > self.tol, xp.int32))
```

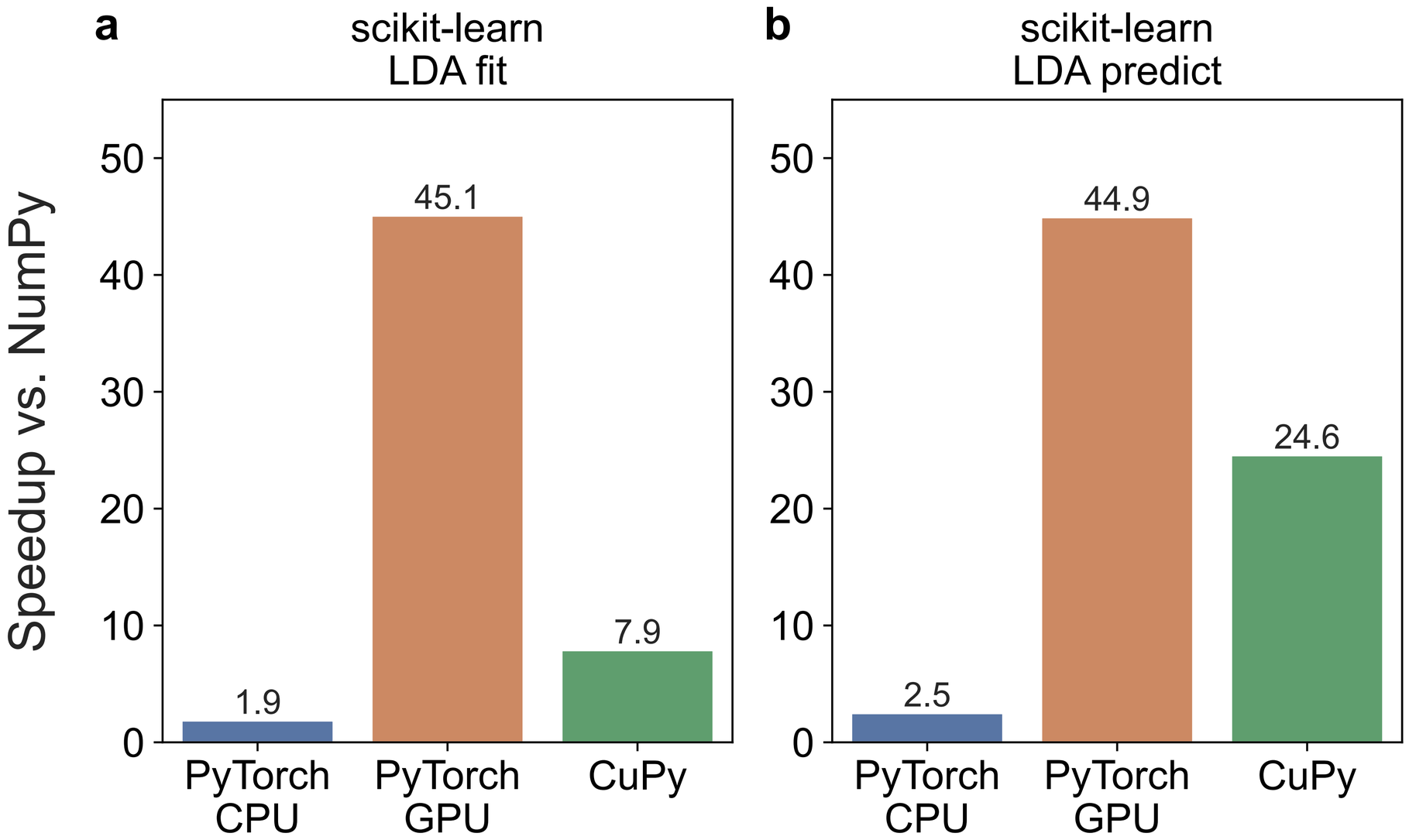
Adoption

- The biggest change is

  `import numpy as np` ⟶ `xp = array_namespace(x)`

  (more on `array_namespace()` later).
- A few minor cleanups from NumPy for portability.
- Most common NumPy functions are standardized.

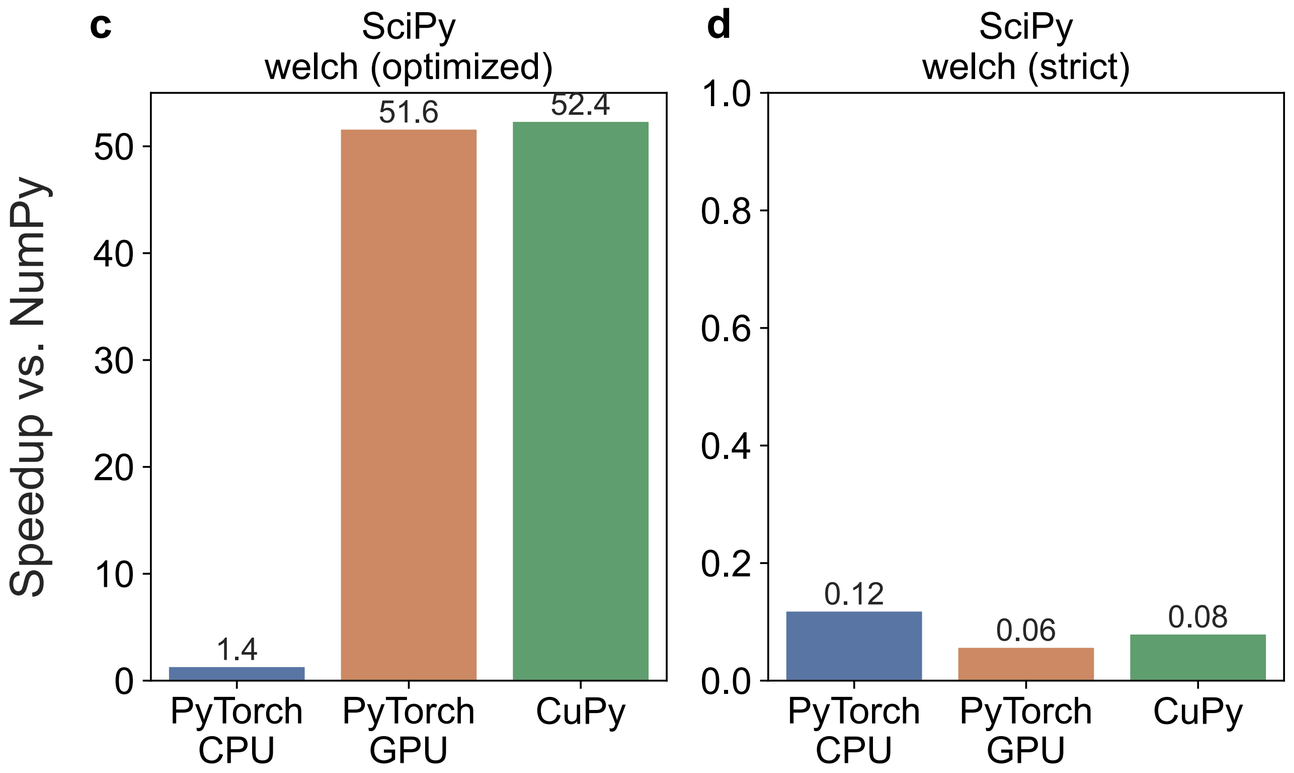
The end result: major speedups vs. NumPy!



Benchmarks were run on Intel i9-9900K and NVIDIA RTX 2080

Adoption

- Occasionally, you may need to special-case a specific array library for performance.
- Example: `scipy.signal.welch()` uses an optimization using array strides.
- Strides are not standardized because some libraries do not expose them (e.g. JAX).
- Strictly array API compliant code is much slower than code that uses strides when available.
- Such library-specific optimizations will be rare and can usually be localized.



Benchmarks were run on Intel i9-9900K and NVIDIA RTX 2080

Tooling

Tooling

We developed several tools to help aid array API adoption:

- Compatibility Layer: `array-api-compat`
- Minimal Implementation: `numpy.array_api`
- Array API Test Suite: `array-api-tests`

Compatibility Layer

- Existing libraries (NumPy, CuPy, PyTorch, etc.) are mostly spec compliant, but not fully.
- `array-api-compat` is a small wrapper around each library to cleanup the small differences.
- Example: `array_api_compat.numpy.concat()` wraps `numpy.concatenate()`.
- Small, venderable, pure Python library with no hard dependencies.

Compatibility Layer

- Example `array-api-compat` usage:

`array_namespace(x)` returns `xp`, the corresponding array API compatible namespace for an array object `x`.

```python
from array_api_compat import array_namespace

def some_function(x, y):
    xp = array_namespace(x, y)

    # Now use xp as the array library namespace
    return xp.mean(x, axis=0) + 2*xp.std(y, axis=0)
```

Minimal Array API Implementation

- numpy.array_api is a strict minimal implementation of the standard
- Fails on any behavior not explicitly required by the standard, even if it is allowed.
- Example:

```
>>> import numpy.array_api as xp
<stdin>:1: UserWarning: The numpy.array_api submodule is
still experimental. See NEP 47.
>>> a = xp.ones((3,), dtype=xp.float64)
>>> b = xp.ones((3,), dtype=xp.int64)
>>> a + b
Traceback (most recent call last):
...
TypeError: float64 and int64 cannot be type promoted together
```

Minimal Array API Implementation

- `numpy.array_api` is not designed for use by end-users.
- Rather, it is for array consuming libraries like SciPy or scikit-learn to test that their implementations are portable against the standard.
- Most aspects of the standard are not strict, e.g., cross-kind type promotion is allowed but not required.
- If your code runs against `numpy.array_api`, it will run against any array API compliant library.

Test Suite

- The standard is quite large (~200 functions, each with a set of specified semantics)
- Need a way to array libraries to test their level of compliance.
- We developed a standard test suite, array-api-tests
- Over 1000 tests for every function and aspect of the standard.
- Has been successfully used to implement compliance in NumPy, CuPy, and PyTorch.
- First example we know of of a library-independent Python test suite.
- https://github.com/data-apis/array-api-tests

Test Suite

Hypothesis (hypothesis.works)

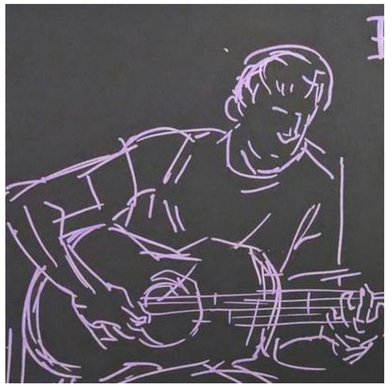
- Makes heavy use of the Hypothesis property-based testing library
- We implemented native Hypothesis support for the array API upstream (`hypothesis.extra.array_api`).
- Hypothesis is a great fit for such a test suite:
  - Property-based tests mean roughly one-to-one translation of the spec to tests.
  - Automatically tests corner cases & all possible combinations.
    - Example: NumPy does not follow type promotion rules, but only for 0-D array + non-0-D array (see NEP 50).

Future Work

- Our focus in 2023 is on adoption.
- NumPy 2.0 (later this year) is planned to have full array API compliance.
- The standard will continue to evolve to match community needs. Ideas for future standardization include
  - device standardization
  - extended data type support (including strings and datetimes)
  - I/O
  - support for mixing array libraries
  - parallelization
  - optional extensions for deep learning and statistical computing.

Future Work

- The Consortium has been working on a similar effort for dataframes
  - Dataframe Interchange Protocol https://data-apis.org/dataframe-protocol/
  - Work-in-progress Dataframe API standard https://data-apis.org/dataframe-api/draft/
    - See the talk by my Quansight colleague Marco Gorelli at EuroSciPy later this year!



Marco Gorelli

Acknowledgements

My Quansight Colleagues

Ralf Gommers
Quansight Labs Co-Director

Thomas J. Fan
scikit-learn core developer

Athan Reines

Pamphile Roy
SciPy core developer

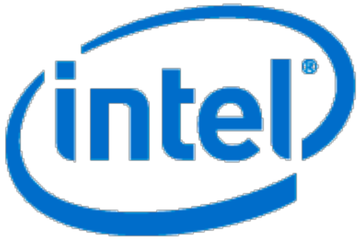
Matthew Barber

Stephannie Jimenez Gacha

Quansight Labs

Acknowledgements

Data APIs Sponsors



Feedback Welcome

- Feedback on the Consortium work is always welcome (new ideas for standardization, requests for help with adoption, general questions, etc.)
  - https://github.com/data-apis/array-api/issues/
- Contributions are also welcome, especially helping various libraries with adoption.
- I will be sprinting on the array API this weekend. If you want help adopting the array API or just have questions, please come!

Questions