High-energy physics analysis in the Python ecosystem

Alex Pearce
alex.pearce@cern.ch

Monday 30th January 2017
DIANA meeting, CERN

What is an analysis?

Means different things to different people and experiments. To me (LHCb):

- Turning a series of observations into an estimation of a physical quantity
- Interesting observables include:
  - Matter-antimatter asymmetries
  - Searches for and characterisation of new and possibly rare processes
  - Production and decay rates
- Almost always looking at an *exclusive* final state
  - Charm meson decay to charged hadrons, e.g. $D^0 \to K^-\pi^+$
  - Semileptonic beauty decays, e.g. $B^0 \to D^{*+}\tau\nu_\tau$
- Fit parametric models to estimate one or two parameters of interest
- Either measure a signal, or set limits

What are the components of an analysis?

- Observations invariably persisted as an *ntuple*, as a ROOT `TTree`
  - Branches are almost always scalar and numerical
  - One branch per physical quantity per particle, e.g. kaon p_{T}
  - Also known as *tabular data*
- Selection optimisation
  - Metrics differ between analyses; often signal purity or relative strength
  - Use rectangular cuts, and/or classification with machine learning
  - Feature engineering, sometimes
- Model building and parameter estimation
  - Fitting PDFs with χ^2 or (un)binned maximum likelihood
  - Simultaneous fit many datasets, and/or many dimensions
  - Tens or > 100 parameters are not uncommon
- Permuting analysis techniques to check for systematic effects
- Statistics!

How does one actually perform an analysis?

1. Write code, about 10 to 100 kLOC
2. Run code, taking from seconds to days
3. `GOTO 1`

- Most of my time is spent writing and debugging code
- This should be a pleasant experience!
- Writing Python makes me happy
- I am continually finding new tools that surprise me and make my life easier
- I write my analyses in Python; it does *almost* everything

Python vs. the world

- Python is fast becoming (has become?) the *lingua franca* of the scientific computing community
  - HEP is starting to realise there is a scientific computing community

Why?

- Simple to learn, quickly become productive
- Legion of well-supported, well-documented packages for many tasks
- "How can I solve this problem?" $\rightarrow$ Google $\rightarrow$ Stack Overflow $\rightarrow$ solution
  - *So* widely used! For everything!
- Python's philosophy is beauty, readability, simplicity, and being explicit
  - This drives many people to write *nice code*
  - `import this`

The Python community

- The hallmarks of a great programming community:
  - Active
  - Open to contributions (from the inside and outside)
  - Committed to a stable, well-documented, sensible user interface (API)
  - Integration with other packages
  - Welcoming of competition
- Many Python packages embody these principals:
  - Data manipulation: Numpy, pandas
  - Algorithms: scipy
  - Plotting: matplotlib, seaborn, Bokeh
  - Machine learning: scikit-learn, TensorFlow
  - Domain specific: astropy, biopython
  - Others: IPython, Flask, requests,

Python packages

Numpy

- Numpy *arrays* are data structures of a homogeneous datatype
- Elements stored in contiguous memory locations
- Operations are performed element-wise, like vector algebra

```
import numpy as np
a = np.array([[0, 2],
              [0, 1]])
print(2*a)
# array([[0, 4],
#        [0, 2]])
```

- Extremely fast
- Pays to understand it (this is generally a Good Idea)
  - Never explicitly write a loop over a numpy array

Python packages

pandas

- pandas `DataFrame` are like `TTree`, but with an explicit (possibly multidimensional) index
- Get the vector of column values with `df.column_name`
- Data stored as numpy arrays behind the scenes
- Same element-wise behaviour

Python packages

pandas

```python
import numpy as np
import pandas as pd

# Generation
data = np.random.normal(loc=20, scale=10, size=100)
# Selections
selection_mask = data > 0
# Pandas
df = pd.DataFrame(dict(pion_pt=data[selection_mask]))
print(df.index)
# RangeIndex(start=0, stop=96, step=1)
print(df.pion_pt.mean(), df.pion_pt.std())
# (20.29, 8.90)
print(df.quantile([0.01, 0.99]))
#          pion_pt
# 0.01    1.951587
# 0.99   41.139315
```

Python packages

pandas

```
df = df.assign(kaon_pt=2*df.pion_pt)
cut = pd.cut(df.pion_pt, [0, 5, 15, 100])
print(cut)
# pandas shows: index, bin boundaries
# 0       (5, 15]
# 1     (15, 100]
#          ...
# 93    (15, 100]
# 94    (15, 100]
print(df.kaon_pt.groupby(cut).mean())
```

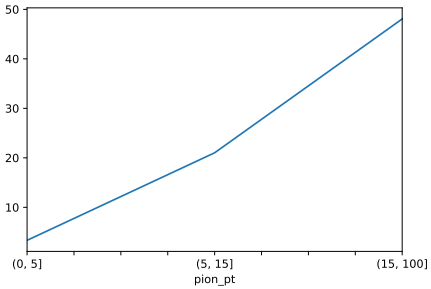
Python packages

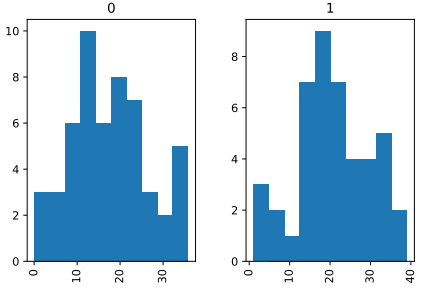
pandas and matplotlib

```
import matplotlib.pyplot as plt

mean_kaon_pt = df.kaon_pt.groupby(cut).mean()
mean_kaon_pt.plot()

df = df.assign(is_background=np.random.choice([0, 1],
                                              size=len(df)))
df.pion_pt.hist(bins=10, by='is_background')
```





Python packages

pandas

```
df = pd.DataFrame(dict(
    pe=...,
    px=...,
    py=...,
    pz=...,
    ...
))
four_vector = df[['pe', 'px', 'py', 'pz']]
pt = np.sqrt(four_vector['px']**2 + four_vector['py']**2)
p_magnitude = np.linalg.norm(four_vector[four_vector.columns[1:]])
# We should have a package that wraps this!
```

Python packages

pandas - usage in HEP

- It's really fast
  - In execution and writing the code
- Try to think in vector operations
  - It's very weird at first
  - Explicitly looping kills performance, don't!
- Easy to run out of memory if you're not thinking
  - Solution 1: don't load every branch
  - Solution 2: Dask, drop-in replacement for 'out-of-core' DataFrame computation
- Go through the documentation to see what's possible
- For loading everything ROOT:
  - `root_numpy` for `TTree`, `TH*`, ... to numpy arrays
  - `root_pandas` for getting a `DataFrame`
  - Useful as things are often stored on EOS, accessible with XRootD

Python packages

matplotlib

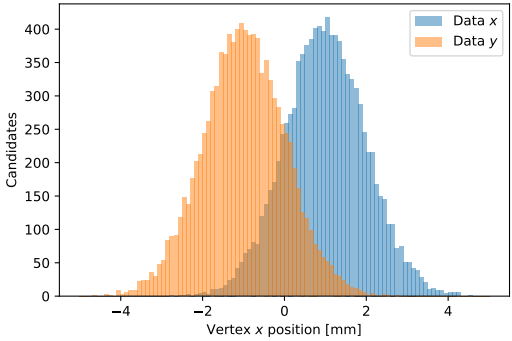
- *The* Python plotting package
- Recently released version 2.0, with greatly improved defaults for pretty plots

```
import matplotlib.pyplot as plt
import numpy as np

data1 = np.random.normal(loc=1, size=10000)
data2 = np.random.normal(loc=-1, size=10000)
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
# Add normed=True for histograms normalised to unit area
plot_kwargs = dict(bins=100, range=(-5, 5), alpha=0.5)
ax.hist(data1, label='Data $x$', **plot_kwargs)
ax.hist(data2, label='Data $y$', **plot_kwargs)
ax.set_xlabel('Vertex $x$ position [mm]')
ax.set_ylabel('Candidates')
ax.legend()
```

Python packages

matplotlib



- The API is sane
- The plots are pretty (see seaborn for ultra-pretty, and ultra-interesting, visualisation)
- I haven't manually adjusted label positioning in forever
- Can't save plots as `.C` files, I guess?

Python packages

scikit-learn

- An exceptionally well-crafted package
- Superb documentation acts as API reference and high-level ML textbook
- Handles both numpy arrays and DataFrames as input formats

```
from sklearn import tree
X = [[0, 0], [1, 1]]
Y = [0, 1]
clf = tree.DecisionTreeClassifier(max_depth=3)
clf = clf.fit(X, Y)
print(clf.predict_proba([[2., 2.]]))
# array([[ 0.,  1.]])
```

Python packages

scikit-learn, hep_ml

Builds on scikit-learn, adding some HEP-specific tools

- Classifier that will try to fit a flat efficiency in a set of features
- 'Histogram reweighter', for modelling likelihood ratios
  - I don't divide histograms any more
- sPlot method for taking per-observation PDF values and returning per-species sWeights
  - Fast, and a very terse implementation

Python packages

Jupyter

- Interactive web page for executing code in a shell-like environment
- Fantastic for debugging and prototyping
- Not ideal for housing 'real', deeply complex analysis code
  - You'll need to write at least one Python module in your analysis
- The code presented here is available in a notebook:

http://nbviewer.jupyter.org/gist/alexpearce/a18fcdb8a13093cac4468b9f341cffa2

'Failures' of the ecosystem

- I think there are some things missing that are often reinvented by analysts
  - Aforementioned four vector class being an example
  - Histogram objects? (see histogrammar)
  - Histogram plotting with error bars (see `missing_hep`, `astropy`)
  - Statistics? Quite a touchy subject sometimes
- Most prominent gap in the market is a flexible and powerful cost function builder to replace RooFit
  - iminuit exists once you have your cost function
  - probfit is OK, but isn't actively maintained
  - tensorprob is a super interesting proof of principle, using TensorFlow to build models (allows multicore and GPU evaluation)
- Whatever we build, it must:
  - Play nice with the existing ecosystem
  - Have a consistent, considerate, *enjoyable* API
    - *Not* orthogonal to writing C/C++
    - See Cython, pybind11

Failures of the our 'ecosystem'

- We're not educating our students with transferable skills
- Performing analyses with Python is great way to get people up to speed with how *everyone else* is doing things
- We should also be teaching good coding practices, how to contribute to software projects (even internal), how to solve problems with code
  - Or provide tools such that students do a much smaller amount of coding than they do now
- We're doing ourselves a disservice by not even *trying* alternative tools, we're curious people, let's try stuff!

Summary

- Python makes analysis fun! (fun not guaranteed)
- So much to cover; do ask me for more information on anything
- I think Python is already suitable for most things I want to do for 'analysis', and probably for you as well
- There is some stuff that's missing, but *we* can build those packages
  - No monoliths!

Backup