Thoughts on using python, numpy, and scikit-learn for HEP analysis

Gordon Watts
University of Washington
Poster #278

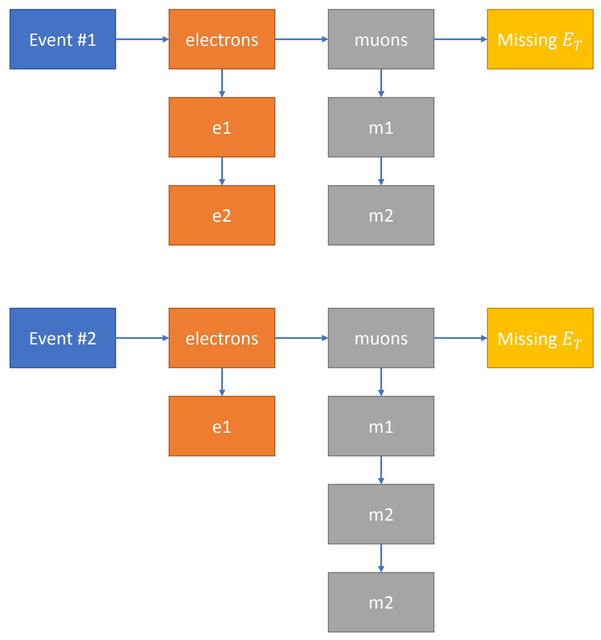
Introduction

What would analysis look like in another language if it wasn't based on C++? If it wasn't based so explicitly on ROOT?

Python and numpy are not yet ready for a full translation

I wanted to base the analysis on the numpy and pandas python packages. Numpy is an open source Python package that contains an ultra efficient N-dimensional array along with mathematical functions to operate on columns and rows of the array. Numpy is implemented in CPython, which is compiled code. Pandas makes dealing with numpy easier by giving you short cuts for operations.

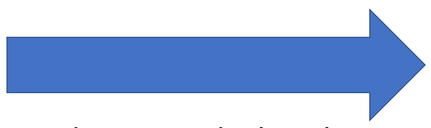
But our data is not well suited for numpy, which is fastest and most at home with an csv-like data format: square arrays or tables.



Event	E1 pT	E2 pT	E3 pT	M1 pT	M2 pT	M3 pT	M ET
1	56.2	25.4	0.0	133.0	78.0	0.0	74.0
2	150.0	0.0	0.0	190.3	110.5	25.0	310.0

Our data does not fit well in a rectangles!

- Pad with zeros?
- Drop "extra" jets?
- Give each event a max number of jets?



I've taken some basic plotting tasks and examined what they look like in various environments. All code can be found on github: https://github.com/gordonwatts/analysis-plot-comparison

Analysis Tasks

The following tasks were setup to look at the performance and user interface for doing late-stage ntuple analysis.

A simple ntuple file with single-value leaves and leaves that contain vector<double>'s. It looks a lot like an end-stage *flat* ntuple that many analyses use. It contains no complex objects (not even a TLoretnzObject). The total file size is 11 GB, and it has 148 branches and 1.9 million events. The tasks:

1. Plot the missing H_T, which is a single-value leaf
2. Plot jet p_T. This is a leaf of vector<double>, so requires iterating over both events and the array.
3. Plot jet p_T with $|\eta| < 1.0$. This requires matching jet p_T to jet η.
4. Plot missing H_T for events that have two good jets. This requires iterating over jets, and making a cut at the event level.

LINQ

Setup & Configuration

This uses the author's library to turn functional C# code into C++ that is compiled and runs against a ROOT file. A special command must be run against the ROOT file in order to generate the types from the leaves (note root leaves appear as C# structure references).

Plotting Missing H_T

This requires two lines:

```
var tree = CreateQueriable("<filename>.root");

using (var f = new FutureTFile(new
           FileInfo("../../../../01.root")))
{
  tree
    .Select(e => e.event_HTMiss)
    .Plot("met", "MET; Missing H_{T} [GeV]",
        100, 0.0, 500.0)
    .Save(f);
}
```

The loop over events is implied. The Select statement isn't really needed – that could have been specified in the Plot function. It is here for clarity.

Plotting Jet p_T

Plotting Jet p_T is different: the sub-array reference is explicit (this is different in other methods):

```
tree
  .SelectMany(e => e.jets)
  .Select(j => j.CalibJet_pT)
  .Plot("pt", "pt; pt [GeV]", 100, 0.0, 500.0)
  .Save(f);
```

The SelectMany takes an array and converts it to a sequence.

Plotting Jet p_T for $|\eta| < 1$

Selection with jet η becomes straight forward once the sub-array sequence above is understood:

```
tree
  .SelectMany(e => e.jets)
  .Where(j => Math.Abs(j.CalibJet_eta) < 1.0)
  .Select(j => j.CalibJet_pT)
  .Plot("pt", "pt; pt [GeV]", 100, 0.0, 500.0)
  .Save(f);
```

Note the filter statement is a lambda. C# allows lambdas to be passed as AST, and the ROOTToLINQ library converts the AST into C++ code.

Plot Missing H_T for Events that have Two Good Jets

For a cut the sub-sequence reference is contained in a single line:

```
tree
  .Where(e =>  e.jets.Where(j =>
     Math.Abs(j.CalibJet_eta) < 1.0
     && j.CalibJet_pT > 40.0).Count() >= 2)
  .Select(e => e.event_HTMiss)
  .Plot("met", "MET; Missing H_{T} [GeV]",
     100, 0.0, 500.0)
  .Save(f);
```

The filter statement contains a complete sub-query applied for each jet, with the count operator finally to sum up the number of jets that satisfy the criteria.

Python with numpy

Setup & Configuration

This requires download and install of Anaconda 3 and pip install of *uproot.* The uproot package is used to read TTree's without having to link against ROOT. Matplotlib is used to make plots.

Plotting Missing H_T

This requires two lines:

```
reco_tree = uproot.open("<filename>.root")["recoTree"]
met = reco_tree.array('event_HTMiss')
plt.hist(met, bins=100)
plt.show()
```

The *array* method reads in the complete event_HTMiss branch as a single numpy array.

Plotting Jet p_T

Plotting Jet p_T is identical. However, what goes on behind the scenes is a little different. Instead of an array, the `reco_tree_array('CalibJet_pT')` call returns a jagged array. This is an array of arrays. For every event there is a list of jets p_T's:

```
jaggedarray([[ 397.33746875  102.14053125  101.37173438   46.37198437   21.0346543 ],
             [ 325.38840625  224.84854688   87.839875   ...  18.348625    18.21963672  17.38869336],
             [ 355.8906875   175.6294375   124.04192187 ...  21.37303516  20.47201172  20.25007031],
             ...,
             [ 383.48553125  311.0424375    80.18246094   26.22417773   18.08358008],
             [ 257.60276563  158.55284375   77.27360156 ...  23.14483984  21.61642773  15.90926465],
             [ 220.57328125   69.04777344   60.0375625  ...  20.88332812  20.22796289  19.06125195]])
```

A jagged array is actually a numpy 1 dimensional array, along with a second array that indexes the start for each event.

Plotting Jet p_T for $|\eta| < 1$

Jet η and p_T are treated as two long, matched, arrays:

```
reco_tree = uproot.open("<filename>.root")["recoTree"]
jetinfo = reco_tree.arrays(['CalibJet_pT',
'CalibJet_eta'])
eta = jetinfo[b'CalibJet_eta']
jetpt = jetinfo[b'CalibJet_pT']
goodeta = np.abs(eta.content) < 1.0
plt.hist(jetpt.content[goodeta], bins=100)
```

The standard trick of creating a mask in numpy here is used to get a list of the jets that satisfy the criteria. Note the use of the ".content" property – this accesses the numpy array that backs the jagged array.

Plot Missing H_T for Events that have Two Good Jets

The code is more complex now as we must relate between events and sub-lists of jets:

```
reco_tree = uproot.open("<filename>.root")["recoTree"]
jetinfo = reco_tree.arrays(['event_HTMiss',
'CalibJet_pT',
   'CalibJet_eta'])
eta = jetinfo[b'CalibJet_eta']
good_eta = np.abs(eta.content) < 2.0
pt = jetinfo[b'CalibJet_pT']
good_pt = pt.content > 40.0
good_jet = uproot.interp.jagged.JaggedArray(good_eta &
good_pt, eta.starts, eta.stops)
good_event = [sum(l) > 2.0 for l in good_jet]
mht = jetinfo[b'event_HTMiss']
plt.hist(mht[good_event], bins=100)
```

Here the problem with the current state of python and jagged arrays is apparent. In particular, the good_jet mask must be converted it back into a jagged array. And the calculation of good_event occurs in python, not in numpy, and thus is quite slow (almost 50 seconds). The code is also hard to read.

ROOT's RDataFrame

Setup & Configuration

This requires the download and install of ROOT. This is a simple click and go.

Plotting Missing H_T

This requires two lines:

```
ROOT::RDataFrame df("recoTree", "<filename>.root");
auto met = df.Histo1D("event_HTMiss");
auto f = new TFile("../01-dataframe.root", "RECREATE");
met->Write();
```

Note that branch names are specified as text strings.

Plotting Jet p_T

Plotting Jet p_T is identical. No need to specify a second loop, the RDataFrame sub-system figures out the implied loop automatically.

Plotting Jet p_T for $|\eta| < 1$

RDataFrame encourages you to think of the JetPt and JetEta TTree leaves as just giant long arrays:

```
ROOT::RDataFrame df("recoTree", "<filename>.root");
auto df_good = df.Define("goodjet",
                    "abs(CalibJet_eta) < 1.0")
                 .Define("goodjet_pt",
                    "CalibJet_pT[goodjet]");
auto jetpt = df_good.Histo1D("goodjet_pt");
jetpt->Write();
```

Note that we define new columns goodjet and goodjet_pt and apply them using the [] notation to filter the CalibJet's. From that we define a new column that are the filtered jets and we can manipulate them. The filter is untyped.

Plot Missing H_T for Events that have Two Good Jets

The code is more complex now as we must relate between events and sub-lists of jets:

```
ROOT::RDataFrame df("recoTree", "<filename>.root");
auto df_good = df.Define("goodjet",
          "abs(CalibJet_eta) < 1.0 && CalibJet_pT > 40.0")
                 .Define("goodjet_pt", "CalibJet_pT[goodjet]")
                 .Filter("goodjet_pt.size() >= 2");
auto met = df_good.Histo1D("event_HTMiss");
met->Write();
```

- Implicit looping over jets
- Implicit counting over per jet, despite syntax being the same
- C++ code as text, compiled by the clang back-end. Not typed.

Conclusion

Speedwise they are similar. The RDataFrame won overall, LINQ was slowed down by compile time, python came in last by a long shot on the missing H_T with 2 good jets query due to the non-numpy loop.

For readability, which is subjective, the LINQ code came in first, with RDataFrame next, and python last. A TTreeReader test was also done (not shown here) and it comes in between LINQ and RDataFrame.

What can we do better as we head into the HL-LHC?