New RooFit PyROOT interfaces for connections with Machine Learning

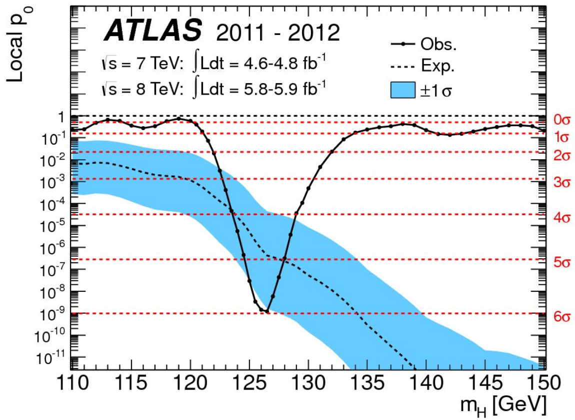
Robin Syring, Jonas Rembser, Lorenzo Moneta

23 October, CHEP 2024

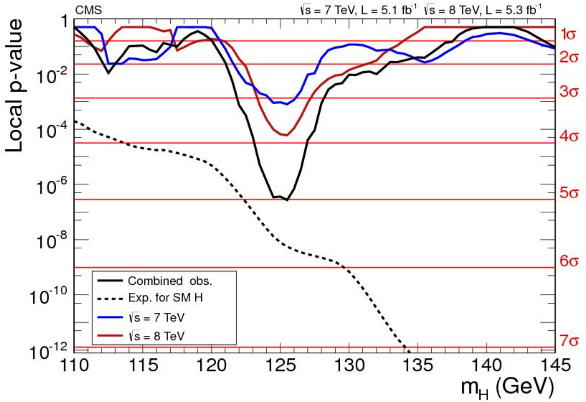
ROOT
Data Analysis Framework
https://root.cern

Introduction to RooFit

- **RooFit**: C++ library for statistical data analysis in ROOT
  - provides tools for model building, fitting and statistical tests
- Recent development focused on:
  - **Performance** boost (preparing for larger datasets of **HL-LHC**)
  - More **user friendly** interfaces and high-level tools



In **this presentation** we're showing how targeted new features like **using Python functions inside RooFit** can unlock the world of **Simulation Based Inference (SBI)** in RooFit



This talk builds on top of RooFit developments shown at **previous conferences**:

- [ACAT 2021 talk] showcasing pythonizations
- [CHEP 2023 talk] presenting new vectorizing RooFit

Simulation Based Inference (SBI)

- In case where you don't have analytic models for probability, but you can **sample** with MC simulators
- Learn **(parametrized) likelihood ratio** to do parameter estimation **without any histograms**

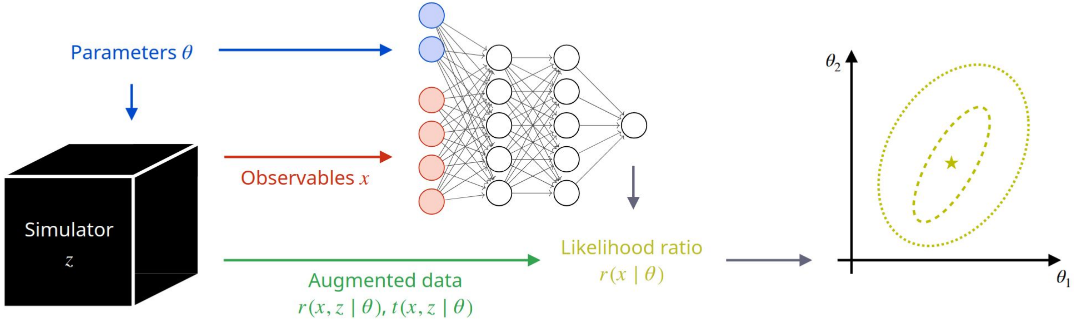


Figure borrowed from [Alexander Held's talk](#) at the [PHYSTAT-SBI 2024 workshop](#)

$$\mathrm{NLL}(\theta) = -\sum_i \log p(x_i|\theta)$$

likelihood ratio trick

$$-\sum_i \log \frac{p(x_i|\theta)}{p_{\mathrm{ref}}(x_i|\theta)}$$

learn likelihood ratio from MC samples

$$-\sum_i \log \frac{s(x_i|\theta)}{1-s(x_i|\theta)}$$

Our Goals

1. **Enable SBI in RooFit** and show tutorial with **most basic example**

2. **Demonstrate** our users how they can avoid **shortcomings of histogram-based strategies** with SBI *(in particular curse of dimensionality)*

3. Create more **advanced example with real LHC data**

4. Spread the word and **gather feedback** to guide future development

The Hello World of SBI - 1D fit with one parameter

- Our "`Hello world`": **Gaussian** with one parameter and uniform reference distribution
- Simple to **sample** from these distributions
  - but don't sample too much, in real life sampling is expensive
- We also have **analytical NLL** for reference
- Implemented in the [rf615 tutorial]()

Tutorial idea:

- MC samples with floating *x* and *mu* from Gaussian and from uniform
- train conditional MLP classifier: *s(x,mu)*
- Create yet another MC sample with fixed *mu*: the "observed data"
- Use classifier score for parameter **inference**

$$\mathrm{NLL}(\mu) = -\sum_i \log \frac{\mathrm{Gaussian}(x_i|\mu,\sigma=0)}{\mathrm{uniform}(x_i)}$$

analytical

SBI

$$\mathrm{NLL}(\mu) = -\sum_i \log \frac{s(x_i|\mu)}{1-s(x_i|\mu)}$$

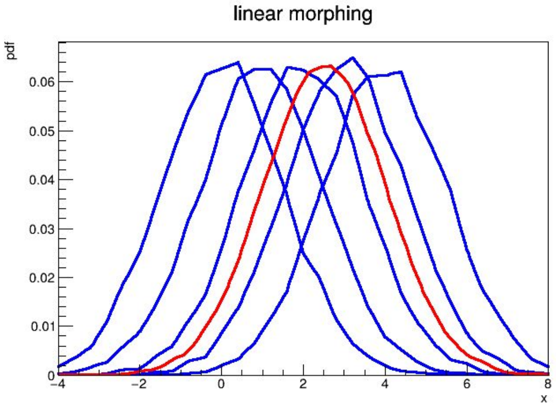
The Hello World of SBI - Results

- We used **40000 MC samples** for training
- Classifier trained naively, no hyperparam. tuning
- Real likelihood ratio **approximated well**
- Compared with traditional **template morphing**:
  - Both SBI and morphing do well

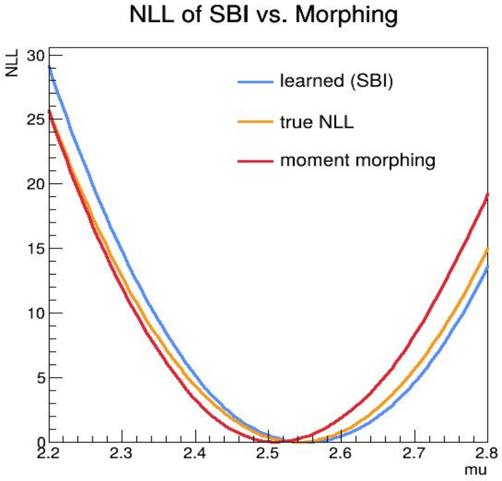
$$\mathrm{NLL}(\mu) = -\sum_i \log \boxed{\frac{s(x_i|\mu)}{1-s(x_i|\mu)}}$$

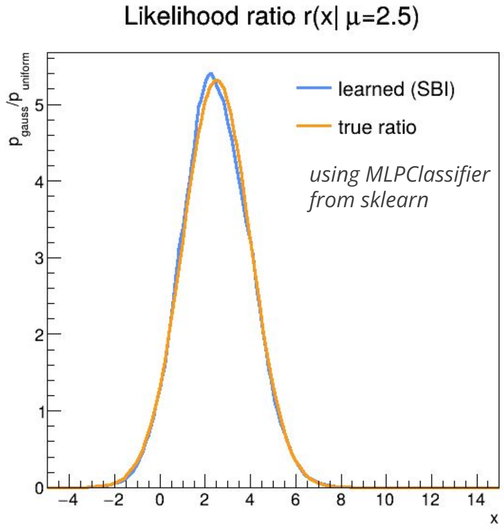
NLL sum over "observed" data

plot with fixed mu for validation



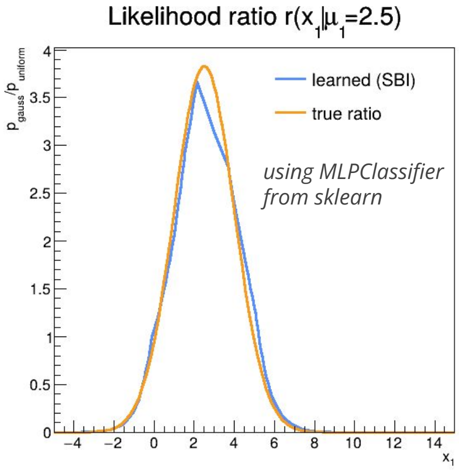
template morphing illustration (see also [this presentation](#))

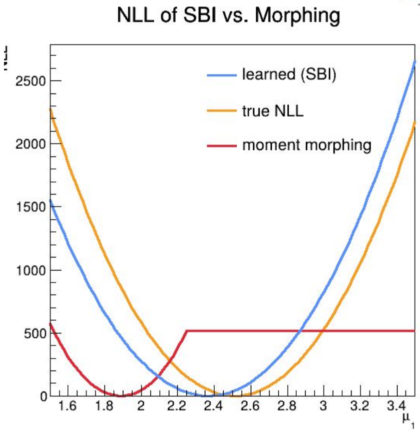




Extending to multiple dimensions

- The [rf617 tutorial] **extends** the previous **example to 2D**:
  - two uncorrelated Gaussians for x1, x1 with params mu1 and mu2
- Everything else the same, also the number of toy MC samples for training (40000 samples)
- **SBI model** has to learn larger phasespace: performance **deteriorates a bit**
- **Template morphing** approach **suffers bigger hit in accuracy** as expected





moment morphing suffers curse of dimensionality!

This confirms that **SBI is very useful** for likelihoods with **many parameters and observables**

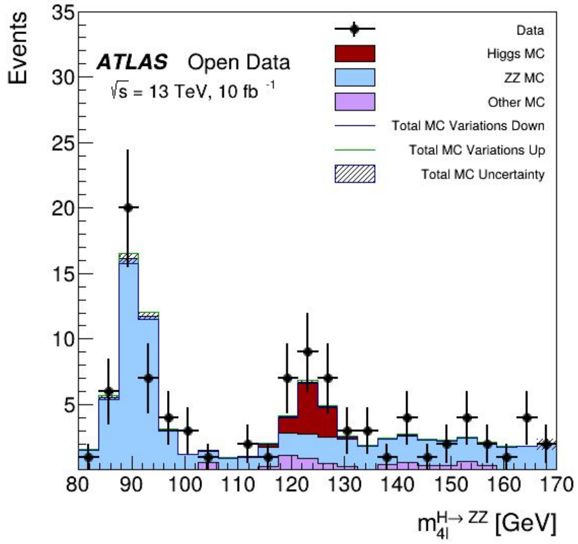
Higgs to four leptons open data example

What about realistic usecases and **real data**?

Usecase: quick **histogram-free** statistical analysis of **Higgs to four leptons** in ATLAS Open Data

- Prediction is given by a **stack of MC samples**
- One observable: *m4l*
- One parameter: scaling of the signal part, aka. ***signal strength mu***

The output of the [RDataFrame tutorial df106](), based on ATLAS Open Data

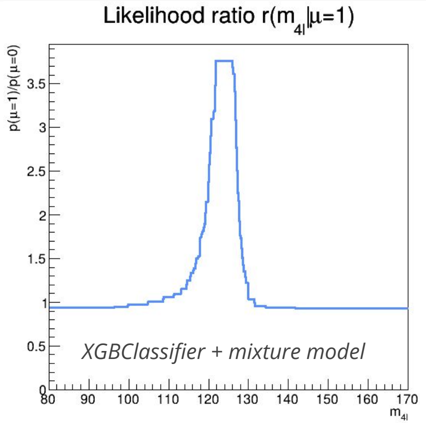


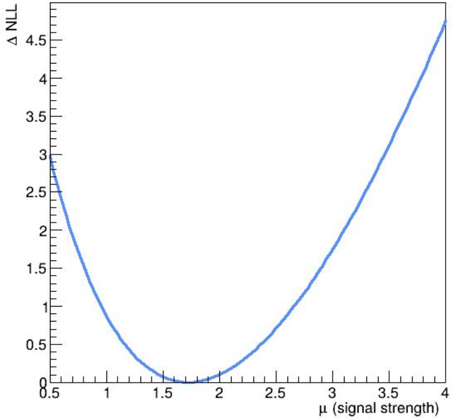
Higgs to four leptons result

- The [rf618 tutorial]() shows this analysis, which f**ollows up on the dataframe tutorial**
  - First RooFit tutorial that **uses open data**!
- The final likelihood ratio is implemented with the **mixture model** formula as a function of signal strength and classifiers to discriminate MC samples
  - Like this, no parametrized classifier is required
- Results agree with what is expected after visually inspecting the histograms

$$
\begin{aligned}
\frac{p(\mathbf{x}|\theta_0)}{p(\mathbf{x}|\theta_1)} &= \frac{\sum_c w_c(\theta_0) p_c(\mathbf{x}|\theta_0)}{\sum_{c'} w_{c'}(\theta_1) p_{c'}(\mathbf{x}|\theta_1)} \\
&= \sum_c \left[\sum_{c'} \frac{w_{c'}(\theta_1)}{w_c(\theta_0)} \frac{p_{c'}(\mathbf{x}|\theta_1)}{p_c(\mathbf{x}|\theta_0)} \right]^{-1} \\
&= \sum_c \left[\sum_{c'} \frac{w_{c'}(\theta_1)}{w_c(\theta_0)} \frac{p_{c'}(s_{c,c'}(\mathbf{x};\theta_0,\theta_1)|\theta_1)}{p_c(s_{c,c'}(\mathbf{x};\theta_0,\theta_1)|\theta_0)} \right]^{-1}
\end{aligned}
$$

mixture model formula from [this paper]()





Vectorized Python functions in RooFit

- RooFit can now wrap Python functions that take and return **NumPy arrays**
- In the Open Data tutorial, this is used twice:
    - wrap the **XGBoost classifier**
    - implement the **mixture model**
- Finally, we pretend to RooFit the likelihood ratio is a **normalized pdf**
- We can then use other RooFit features, like **extended likelihood fits**

```
# Set up RooRealVars before: m4l, mu, n_sig, n_bkg

def llr_zz_vs_higgs_f(m4l: np.ndarray)  -> np.ndarray:
    prob = model_xgb.predict_proba(m4l.T)[:,  0]
    return (1 - prob) / prob

def mixture_model_f(llr: np.ndarray, mu: np.ndarray)  -> np.ndarray:
  return ... # some numpy code (note that mu is 1D ndarray)

llr_zz_vs_higgs = RooFit.bindFunction( "llr_zz_vs_higgs" , llr_zz_vs_higgs_f,
m4l)
llr_mixture = RooFit.bindFunction( "llr_mixture" , llr_mixture_model_f, llh, mu)

pdf = RooWrapperPdf( "pdf", "", llr_mixture,  selfNormalized =True)

# better do extended fit
n_pred = RooFormulaVar( "n_pred", "n_bkg + mu * n_sig" , [mu, n_sig, n_bkg])
pdf_extended = RooExtendPdf( "pdf_extended" , "", pdf, n_pred)


nll = pdf_extended .createNLL(data)
```

trick to bypass auto-normalization

Useful pythonizations for these workflows

Which **PyROOT features** enabled these workflows?

- **Callbacks to Python from C++ code** in PyROOT, preferably done either by:
  - `std::function<T>` pythonization
  - virtual dispatching by **inheriting from C++ class** in Python
- *Note: implementing callback mechanisms via the CPython API is more error prone*
- Copy-free data transfer between C++ and Python:
  - *Python to C++:* Implicit conversion from **NumPy arrays to C-style arrays**
  - *C++ to Python:* **Python buffer interface** support for C-style arrays
  - *See backup for example*

Step up your own interoperability game with this tech!

```python
# Demo 1: std::function pythonization

ROOT.gInterpreter.Declare( """

int myfunc(std::function<int(int)> func) {
  return func(2);
}
""")
print(ROOT.myfunc(lambda x: x * x))


# Demo 2: C++ virtual dispatching from Python classes

ROOT.gInterpreter.Declare( """
class MyBaseClass {
public:
  void talk() { std::cout << getSpeech() << std::endl; }
  virtual std::string getSpeech() { return "I'm base!"; }
};
""")

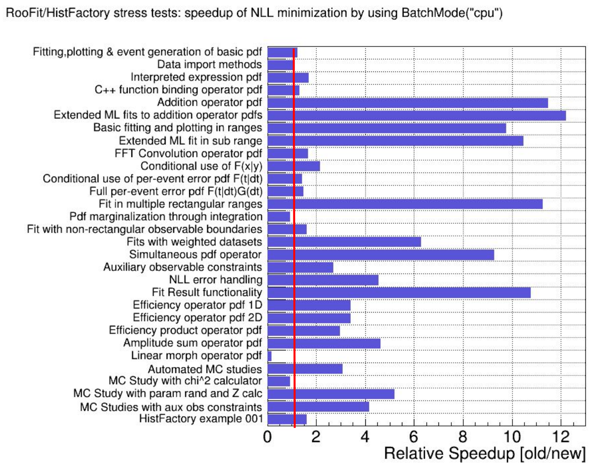
class MyDerivedClass(ROOT.MyBaseClass):
   def getSpeech(self):
       return "I'm derived in Python!"

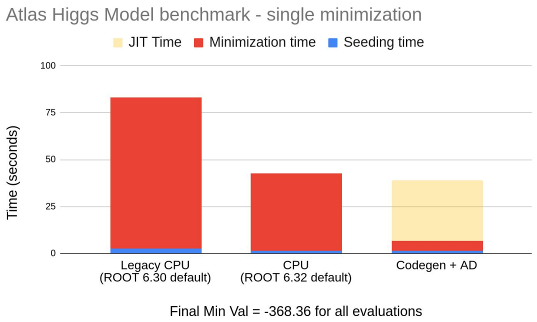
MyDerivedClass().talk()
```

RooFits vectorized evaluation interface

- New **vectorizing RooFit evaluation interface**: presented at previous conferences, provides **great speedup**, the **default since ROOT 6.32**
- Requires implementing this method in your **RooFit class**, which fills computation result into context object:
  - ```
    void RooAbsReal::doEval(
        RooFit::EvalContext & ctx
    )
    ```
- This is used together with C++ virtual dispatching from Python to implement our usecase:
  - `RooPyBind`: C++ class that implements what RooFit requires and has a new **virtual evaluation method** for intended override in **Python**
  - **Pythonization** of `RooFit::bindFunction` does the rest

Without interface for vectorized evaluation, the SBI integration **would not have been possible**.




    ```

Our efforts to be more inviting for developers

We want to make **contributing to RooFit**'s C++ and Python code as **easy as possible**:

- Standalone RooFit build on top of existing ROOT installation
- Workflow to develop RooFit pythonizations without having to build any part of the ROOT CMake project

Conclusions and outlook

- New pythonizations allow you to **wrap Python functions that work with NumPy arrays inside RooFit**

- Main intended use: bring **ML models trained with Python libraries** inside your RooFit model to do **neural simulation based inference**

- New tutorials show this for three examples of increasing complexity:
  - 1D Gaussian fit with one parameter and Multidimensional Gaussian fit
  - **Mixture model** fit to **open Higgs to four leptons data**

- Many possible ways to continue based on eventual **user demand**:
  - **New RooFit classes** for operations with neural likelihood ratios *(like mixture model)*?
  - Support specific usehases like **EFT analysis**?
  - Enable **serialization** of SBI models with RooWorkspace?

*This is mostly **new territory**, easy for early adopters and contributors to **make an impact**!*

Backup - Data transfer between Python and C++ with NumPy arrays

```python
ROOT.gInterpreter.Declare( """

class Squarer {
public:
  Squarer(std::size_t n) : fBuffer(n)  {}

  double * call(double * x) {
     for (std::size_t i = 0; i < fBuffer.size(); ++i) {
        fBuffer[i] = x[i] * x[i];
     }
     return fBuffer.data();
  }

private:
  std::vector<double> fBuffer;
};

""")

arr = np.array([ 1., 2., 3.], dtype=np.float64)

squarer = ROOT.Squarer( len(arr))

# Pass NumPy array, and also create new NumPy array from output.
# Conversions are zero-copy operations!
arr_square = np.frombuffer(squarer.call(arr),  dtype=np.float64, count=len(arr))

print(arr_square)
```