Simple ML Tutorial

Mike Williams
MIT

June 16, 2017

Machine Learning

ROOT provides a C++ built-in ML package called TMVA (T, since all ROOT objects start with T, and MVA for multivariate analysis).

TMVA is super convenient and “comfortable” for people used to using ROOT; therefore, early ML usage in HEP was mostly TMVA-based. These days, our field is moving more towards more widely used packages like scikit-learn (sklearn), Keras, etc., which provide easy-to-use Python APIs — and are used/contributed to by huge communities (excellent documentation, support, etc — see scikit-learn.org).

Today, I’m assuming that this group is more comfortable with ROOT/C++ than Python, so I’ll do a quick demo in TMVA. The main strategies, ease of use, etc, are shared with Python packages like sklearn.

N.b., see also pypi.python.org/pypi/hep_ml/0.2.0 and https://github.com/yandex/rep for HEP-ML tools (e.g. ROOT TTree to numpy array conversion, HEP-specific algorithms, etc).

TMVA Demo

TMVA comes with many built-in tutorials, but they are (IMHO) mostly “too nice” for a simple first go, so I wrote this one:

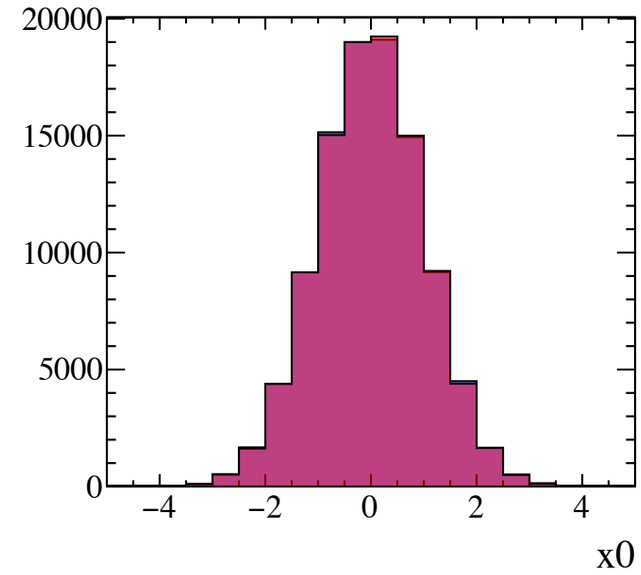
https://www.dropbox.com/sh/o31fb60lzeev96s/AABSRjeQ0vGtm1OSAbI-z93ua?dl=0

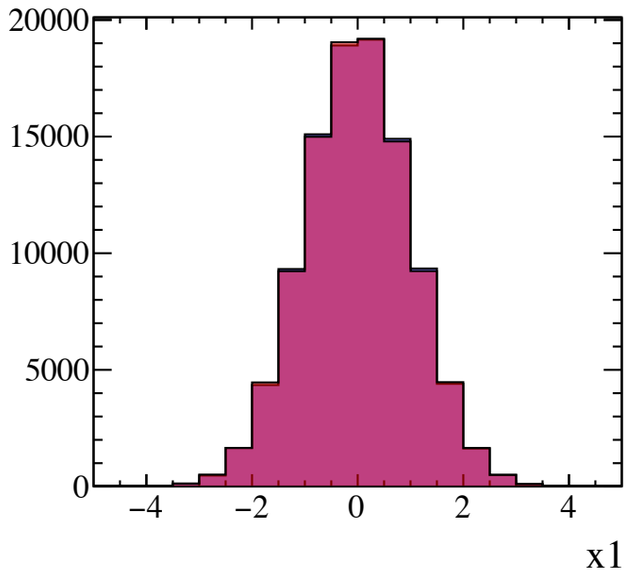
Please download tmvaex.tgz, then do:

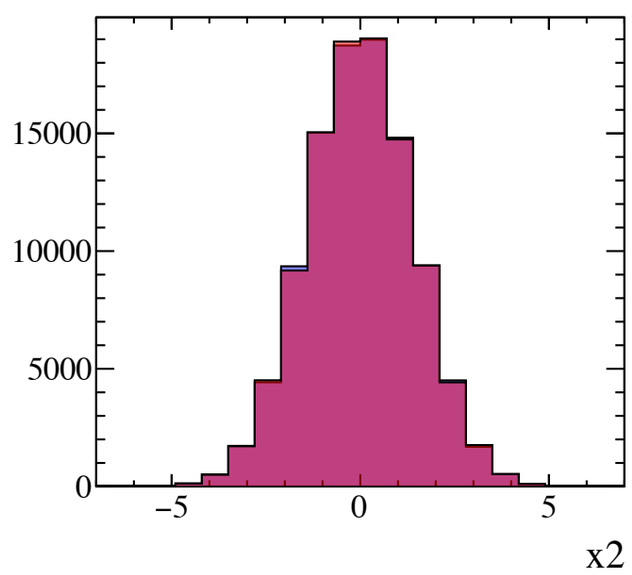
> tar -xzvf tmvaex.tgz
> root
root [0] .L data.C
make signal and background samples (data.1.root and data.0.root)
root [1] makeAll()
plot the features (just to get a feel for the toy problem)
root [2] plot()

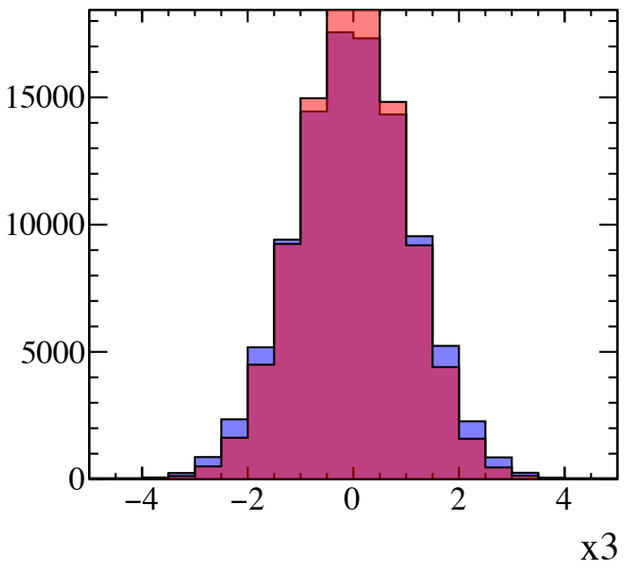
You should see that there is no separation in 1-D in most of the features, and very little in the others. The difference between the two PDFs is in their higher-D correlations — which is where ML is most useful.

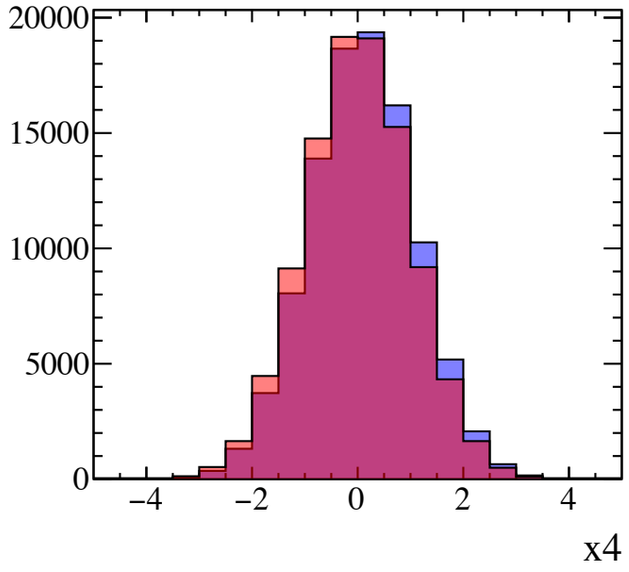
Plots

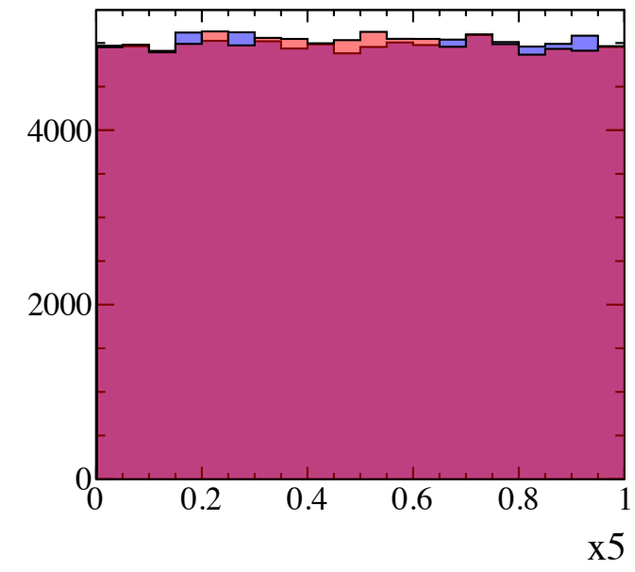

20000
15000
10000
5000
0
−4
−2
0
2
4
x0

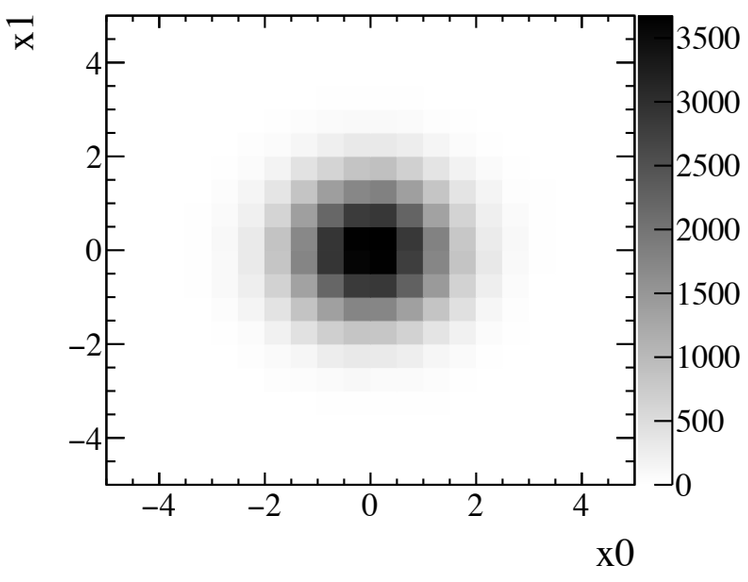

20000
15000
10000
5000
0
−4
−2
0
2
4
x1

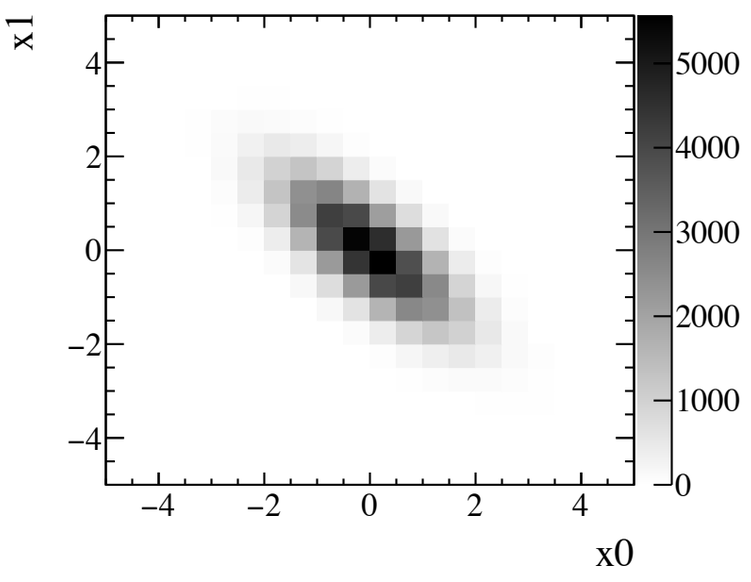

15000
10000
5000
0
−5
0
5
x2


15000
10000
5000
0
−4
−2
0
2
4
x3


20000
15000
10000
5000
0
−4
−2
0
2
4
x4


4000
2000
0
0
0.2
0.4
0.6
0.8
1
x5


x1
4
2
0
−2
−4
−4
−2
0
2
4
x0
3500
3000
2500
2000
1500
1000
500
0


x1
4
2
0
−2
−4
−4
−2
0
2
4
x0
5000
4000
3000
2000
1000
0

Train AdaBoost BDT

First, let's train the classic BDT using AdaBoost and "only" 100 trees and see how we do. Continuing in ROOT:

root [3] .L train.C
the uncommented training string is for AdaBoost with 100 trees, so just run
root [1] train()

Was that faster than you expected? (It will take longer for more trees or a neural network, but not much.)

Scroll up and check out the variable ranking. Does it make sense? (Always check the final one of these, as sometimes it lists them at earlier stages in the learning too.)

Train AdaBoost BDT

My dumb example writes the testing results to tmp.root, and then starts up a GUI with these loaded by calling TMVA::TMVAGui(“tmp.root") — note that you do not need to start up a GUI if you don’t want one.

Let’s play with this:
>Click on the 1a button to have TMVA plot the features, just as a sanity check that you’ve configured things properly.

>Click on the 4b button to see the 1-D response for both data type, which gives a feel for the separation power. Also, compare the distributions for each type from the training and validation samples. This gives a feel for how much overtraining there is.

>Click on 5b to see the ROC curve, e.g., I get 90% background rejection at 70% signal efficiency.

Train AdaBoost BDT

Now, change NTrees=100 to NTrees=1000 (1000 trees) in the string, and rerun the training (.L train.C; train();). This takes 10x longer — but does much better! I now get 90% background rejection with 95% signal efficiency.

This was the same training data and algorithm type, but a change to one hyper parameter. How do we know what these should be set to?

You can get a feel for some broad ranges that are “sensible” by knowing what the parameters do, and by checking the default values in various packages; however, to get the optimal values requires trial-and-error. It is a black-box optimization problem.

Since this is a problem for everybody—and everybody uses ML now—there are really nice packages that will do this optimization for you. E.g., see Spearmint on GitHub: https://github.com/HIPS/Spearmint, which uses Bayesian optimization to quickly find the optimal set of hyper parameters automatically (will take O(10)xN(pars) trainings to find it).

See http://tmva.sourceforge.net/optionRef.html for TMVA parameters.

Train BDTG and Neural Network

OK, now let's try another algorithm. First, if you want, copy tmp.root to bdt.ada.root if you want to compare results later without rerunning.

Comment out the first BookMethod and uncomment the second. This will change the boosting algorithm from AdaBoost to Gradient boost. Now rerun the training (.L train.C; train();). I get similar results, but the point is that using a different method is trivial (copy tmp.root to bdt.gb.root if you want).

Now let's do an MLP neural network. Comment out the previous BookMethod and uncomment the 3rd one. Rerun the training (.L train.C; train();) and now you've trained a single-hidden-layer NN. It takes a bit longer than the BDTs, but not much.

N.b., true Deep Learning, with many more hidden layers, is extremely powerful but also takes a lot more CPU (often many days on a single multicore machine) to train and memory to store the result. For now, this option is expensive, but with TPUs, etc, industry is working hard on making this feasible even for everyday applications in the near future.

See here https://root.cern.ch/doc/v608/TMVAClassification_8C.html for more algorithms in TMVA.

TMVA

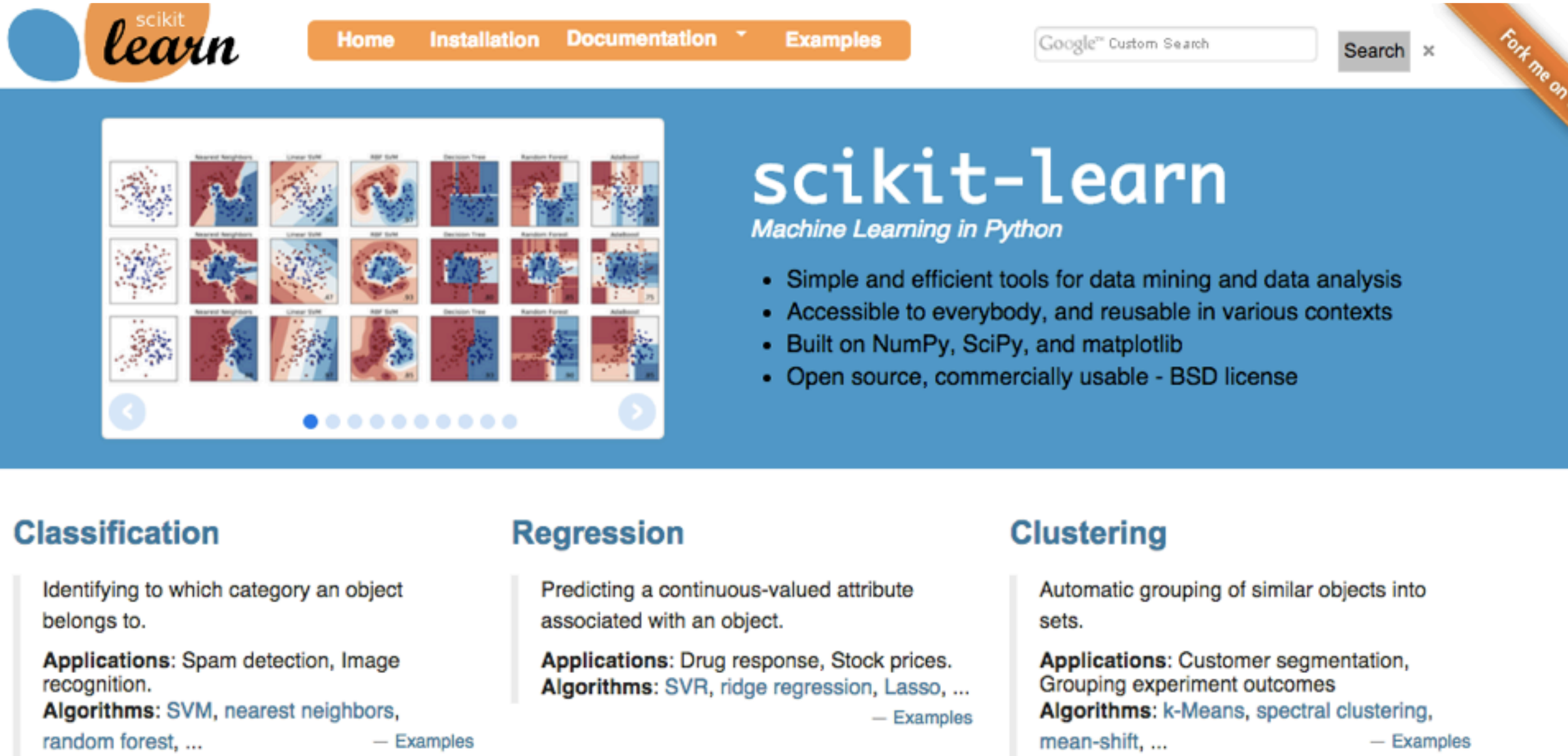
TMVA creates some files locally that contain the info required to run your trained algorithm later. See dataset/weights directory:

The .xml files are used by TMVAs Reader class which you can then pass a set of features and it will compute the response.

The .C files are stand-alone C++ that you can run without even linking to ROOT. You can easily use these files to evaluate the response later.

Tools

Physicists used to mostly use TMVA in ROOT; however, the rest of the world is using the python scikit-learn package (sklearn for short), Keras, etc., and our field is also moving this way.



Basics: Adaboost DT or Multilayer Perceptron NN (MLP); State-of-the-Art: XGBoost DT or Deep NN (e.g. Tensorflow).

Tools, etc.

•ROOT's TMVA is very convenient for physicists, but many are now migrating more and more to scikit-learn, Keras, etc.; i.e., we are moving away from physics-specific software and towards the tools used by the wider ML community. Hyper-parameter tuning using spearmint, hyperopt, etc. (see also Ilten, MW, Yang [1610.08328]).

•Custom loss functions, e.g., response is de-correlated from some set of features (Stevens, MW [1305.7248]; Rogozhnikova, Bukva, Gligorov, Ustyuzhanin, MW [1410.4140]). Already used in several papers (e.g. LHCb, PRL 115 (2015) 161802), and currently being used in many papers to appear soon.

•Many useful tools provided in the HEP-ML package pypi.python.org/pypi/hep_ml/0.2.0, which is basically a wrapper around sklearn, and in REP https://github.com/yandex/rep (both produced by our colleagues at Yandex).

•N.b., beware of non-general optimizations in some algorithms (e.g. CNNs), i.e. make sure to use the right tool for your job.

•Always possible to squeeze out a bit more performance (stacking, blending, etc).

Questions?