Table of Contents

Jet identification in high pile-up environment

Introduction

The identification of jets from pile-up relies on three types of propeties of the jets:

- inside the tracker acceptance, the trajectories of tracks associated to the jets can be used to establish the compatibility of the jet with the primary interaction vertex;
- the topology of the jet shape can be used in order to disentangle jets arising from the overlap of multiple interactions from truly hard jets.
- the object mujltiplicity can be used as an additional handle.

A set of discriminating variables have been developed to quantify such properties:

beta	
RMS (or dR2Mean)	
jet dR profile	
charged and neutral multiplicities	

The strategy to combine the above information is based on a three level approach:

- a cut-based selection explits the minimal amount of information in the simplest way;
- a more sophisticated approach combined all the discriminating variables in a Boosted Decision Tree. Two different BDTs are provided: * A first BDT, labelled as **simple** only explits beta and the jet profile.
- A second BDT, labelled **full** also integrates the information from the multiplicities.

For each of the three types of algorithms, three working points are defined, *loose*, *medium* and *tight*.

Recommendation for:

The Summer16MC contains the latest training of the PU jet ID based on 80X.

How to update JEC for (prescription based on and 2016 data):

The Moriond 2017 MC (CMSSW>=8_0_20) contains the latest training of the PU jet ID based on 80X. To rerun the PU jet ID on the 2016 data with the recommended JEC, cherry-pick the following commit into your CMSSW>=8_0_20 release:

```
git remote add ahinzmann git@github.com:ahinzmann/cmssw.git
git fetch ahinzmann PUidMiniAODfix80
git cherry-pick ca33756e1747aec27d13971bcfd0874b16724e7f
```

An example how to re-calculate the pileup jet ID and re-calibrate the jets on MiniAOD is below:

Show Hide

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("PATUPDATE")
process.load("FWCore.MessageLogger.MessageLogger_cfi")

process.source = cms.Source("PoolSource",
```

```
  fileNames = cms.untracked.vstring(["/RelValTTbar_13/CMSSW_8_0_5-
PU25ns_80X_mcRun2_asymptotic_2016_miniAODv2_v0_gs71xJecGT-v1/MINIAODSIM"])
)

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(10000) )

process.load("Configuration.StandardSequences.FrontierConditions_GlobalTag_cff")
from Configuration.AlCa.GlobalTag import GlobalTag
process.GlobalTag = GlobalTag(process.GlobalTag, '80X_mcRun2_asymptotic_2016_miniAODv2')

process.load("RecoJets.JetProducers.PileupJetID_cfi")
process.pileupJetIdUpdated = process.pileupJetId.clone(
  jets=cms.InputTag("slimmedJets"),
  inputIsCorrected=True,
  applyJec=True,
  vertexes=cms.InputTag("offlineSlimmedPrimaryVertices")
  )
print process.pileupJetId.dumpConfig()

process.load("PhysicsTools.PatAlgos.producersLayer1.jetUpdater_cff")
process.patJetCorrFactorsReapplyJEC = process.updatedPatJetCorrFactors.clone(
  src = cms.InputTag("slimmedJets"),
  levels = ['L1FastJet', 'L2Relative', 'L3Absolute'] )
process.updatedJets = process.updatedPatJets.clone(
  jetSource = cms.InputTag("slimmedJets"),
  jetCorrFactorsSource = cms.VInputTag(cms.InputTag("patJetCorrFactorsReapplyJEC"))
  )
process.updatedJets.userData.userFloats.src += ['pileupJetIdUpdated:fullDiscriminant']

process.p = cms.Path( process.pileupJetIdUpdated + process.patJetCorrFactorsReapplyJEC + process.

process.out = cms.OutputModule("PoolOutputModule",
    fileName = cms.untracked.string("patTupleUpdatedFromMiniAOD.root"),
    outputCommands = cms.untracked.vstring('keep *')
    )

process.endpath = cms.EndPath(process.out)
```

An example how to read the re-ran pileup jet ID from MiniAOD in Python is below:

Show ▸ Hide ▾

```
# import ROOT in batch mode
import sys
oldargv = sys.argv[:]
sys.argv = [ '-b-' ]
import ROOT
ROOT.gROOT.SetBatch(True)
sys.argv = oldargv

# load FWLite C++ libraries
ROOT.gSystem.Load("libFWCoreFWLite.so");
ROOT.gSystem.Load("libDataFormatsFWLite.so");
ROOT.AutoLibraryLoader.enable()

# load FWlite python libraries
from DataFormats.FWLite import Handle, Events
from PhysicsTools.HeppyCore.utils.deltar import deltaR

jets, jetLabel = Handle("std::vector<pat::Jet>"), "slimmedJets"


# open file (you can use 'edmFileUtil -d /store/whatever.root' to get the physical file name)
events = Events("file:////afs/cern.ch/user/h/hinzmann/workspace/tmp/MiniAOD.root")
```

```
for iev,event in enumerate(events):
    #if iev >= 10: break
    event.getByLabel(jetLabel, jets)

    print "\nEvent %d: run %6d, lumi %4d, event %12d" % (iev,event.eventAuxiliary().run(), event.

    # Jets (standard AK4)
    for i,j in enumerate(jets.product()):
        if j.pt() < 20: continue
        print "jet %3d: pt %5.1f (raw pt %5.1f, matched-calojet pt %5.1f), eta %+4.2f, pileup mva
            i, j.pt(), j.pt()*j.jecFactor('Uncorrected'), j.userFloat("caloJetMap:pt"), j.eta(),
        print "Passes loose: ", bool(j.userInt("pileupJetIdUpdated:fullId") & (1 << 2)), "medium:
```

Information on the performance can be found here:

- https://indico.cern.ch/event/559594/contributions/2257924/attachments/1317046/1973307/PUID_JMAR_2016

The data validation and scale factors for 2015 data are documented in AN-16-248. Validation on 2016 data is ongoing.

A preliminary training with 80X MC was also available as userfloat in 80X MiniAODv2, however, the new training described above is expected to give better overall performance.

The 80X MiniAODv1 contains a PU jet ID training based on 76X and you may want to rerun a new pileup jet ID training.

Information for 13TeV data analysis in 76X:

A preliminary 13 TeV training with 76X MC has been completed.

Information on the performance can be found here:

- https://indico.cern.ch/event/502737/contributions/2012692/attachments/1234291/1816811/PileupJetID_76X.pd

The recipe to install the corresponding code is below:

```
cmsrel CMSSW_7_6_4
cd CMSSW_7_6_4/src/
cmsenv
git cms-init
git cms-merge-topic jbrands:pileupJetId76X
git cherry-pick 7c23237a87181e9320874c6f1f913a43fe849499
cd RecoJets/JetProducers/data/
wget https://github.com/jbrands/RecoJets-JetProducers/raw/3dad903ed25d025f68be94d6f781ca957d6f86a
wget https://github.com/jbrands/RecoJets-JetProducers/raw/3dad903ed25d025f68be94d6f781ca957d6f86a
wget https://github.com/jbrands/RecoJets-JetProducers/raw/3dad903ed25d025f68be94d6f781ca957d6f86a
wget https://github.com/jbrands/RecoJets-JetProducers/raw/3dad903ed25d025f68be94d6f781ca957d6f86a
cd ../../..
scram b -j5
```

Pileup jet ID calculator can recalibrate jets with JEC if necessary. Therefore it is important to load the correct global tag or DB file with the latest corrections (example below).

An example how to calculate the pileup jet ID and re-calibrate the jets on MiniAOD is below:

Show Hide

```
import FWCore.ParameterSet.Config as cms
```

```
process = cms.Process("PATUPDATE")
process.load("FWCore.MessageLogger.MessageLogger_cfi")

process.source = cms.Source("PoolSource",
  fileNames = cms.untracked.vstring(["/store/relval/CMSSW_7_6_2/RelValZMM_13/MINIAODSIM/PU25ns_76
#  fileNames = cms.untracked.vstring(["root://xrootd.unl.edu//store/relval/CMSSW_7_6_2/RelValTTba
)

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(10000) )

process.load("Configuration.StandardSequences.FrontierConditions_GlobalTag_cff")
from Configuration.AlCa.GlobalTag import GlobalTag
process.GlobalTag = GlobalTag(process.GlobalTag, '76X_mcRun2_asymptotic_RunIIFall15DR76_v1')

process.load("RecoJets.JetProducers.PileupJetID_cfi")
process.pileupJetIdUpdated = process.pileupJetId.clone(
  jets=cms.InputTag("slimmedJets"),
  inputIsCorrected=True,
  applyJec=True,
  vertexes=cms.InputTag("offlineSlimmedPrimaryVertices")
  )
print process.pileupJetId.dumpConfig()

process.load("PhysicsTools.PatAlgos.producersLayer1.jetUpdater_cff")
process.patJetCorrFactorsReapplyJEC = process.updatedPatJetCorrFactors.clone(
  src = cms.InputTag("slimmedJets"),
  levels = ['L1FastJet', 'L2Relative', 'L3Absolute'] )
process.updatedJets = process.updatedPatJets.clone(
  jetSource = cms.InputTag("slimmedJets"),
  jetCorrFactorsSource = cms.VInputTag(cms.InputTag("patJetCorrFactorsReapplyJEC"))
  )
process.updatedJets.userData.userFloats.src += ['pileupJetIdUpdated:fullDiscriminant']

process.p = cms.Path( process.pileupJetIdUpdated + process.patJetCorrFactorsReapplyJEC + process.

process.out = cms.OutputModule("PoolOutputModule",
    fileName = cms.untracked.string("patTupleUpdatedFromMiniAOD.root"),
    outputCommands = cms.untracked.vstring('keep *')
    )

process.endpath = cms.EndPath(process.out)
```

An example how to calculate the pileup jet ID on AOD is below:

Show Hide

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("PATUPDATE")
process.load("FWCore.MessageLogger.MessageLogger_cfi")

process.source = cms.Source("PoolSource",
  fileNames = cms.untracked.vstring(["/store/relval/CMSSW_7_6_2/RelValZMM_13/GEN-SIM-RECO/PU25ns_
)

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(100) )

process.load("Configuration.StandardSequences.FrontierConditions_GlobalTag_cff")
from Configuration.AlCa.GlobalTag import GlobalTag
process.GlobalTag = GlobalTag(process.GlobalTag, '76X_mcRun2_asymptotic_RunIIFall15DR76_v1')

process.load("RecoJets.JetProducers.PileupJetID_cfi")
process.pileupJetId.jets=cms.InputTag("ak4PFJetsCHS")
process.pileupJetId.inputIsCorrected=False
process.pileupJetId.applyJec=True
```

```
process.pileupJetId.vertexes=cms.InputTag("offlinePrimaryVertices")
print process.pileupJetId.dumpConfig()

process.p = cms.Path(process.pileupJetId)

process.out = cms.OutputModule("PoolOutputModule",
    fileName = cms.untracked.string("patTuplePuIdFromAOD.root"),
    outputCommands = cms.untracked.vstring('keep *')
    )

process.endpath = cms.EndPath(process.out)
```

An example how to read the re-calculated the pileup jet ID from MiniAOD in Python is below:

Show Hide

```
# import ROOT in batch mode
import sys
oldargv = sys.argv[:]
sys.argv = [ '-b-' ]
import ROOT
ROOT.gROOT.SetBatch(True)
sys.argv = oldargv

# load FWLite C++ libraries
ROOT.gSystem.Load("libFWCoreFWLite.so");
ROOT.gSystem.Load("libDataFormatsFWLite.so");
ROOT.AutoLibraryLoader.enable()

# load FWlite python libraries
from DataFormats.FWLite import Handle, Events
from PhysicsTools.HeppyCore.utils.deltar import deltaR

jets, jetLabel = Handle("std::vector<pat::Jet>"), "updatedJets"
pujetids, pujetidLabel = Handle("edm::ValueMap<StoredPileupJetIdentifier>"), "pileupJetIdUpdated"
pujetidDiscriminant, pujetidDiscriminantLabel = Handle("edm::ValueMap<float>") , "pileupJetIdUpda
pujetidFullId, pujetidFullIdLabel = Handle("edm::ValueMap<int>") , "pileupJetIdUpdated:fullId"

# open file (you can use 'edmFileUtil -d /store/whatever.root' to get the physical file name)
events = Events("file:////afs/cern.ch/user/h/hinzmann/workspace/tmp/patTupleUpdatedFromMiniAOD.ro

#eff_loose=0
#eff_medium=0
#eff_tight=0
#num_jets=0

for iev,event in enumerate(events):
    #if iev >= 10: break
    event.getByLabel(jetLabel, jets)
    event.getByLabel(pujetidLabel, pujetids)
    event.getByLabel(pujetidDiscriminantLabel, pujetidDiscriminant)
    event.getByLabel(pujetidFullIdLabel, pujetidFullId)

    print "\nEvent %d: run %6d, lumi %4d, event %12d" % (iev,event.eventAuxiliary().run(), event.

    # Jets (standard AK4)
    for i,j in enumerate(jets.product()):
        if j.pt() < 20: continue
        print "jet %3d: pt %5.1f (raw pt %5.1f, matched-calojet pt %5.1f), eta %+4.2f, pileup mva
            i, j.pt(), j.pt()*j.jecFactor('Uncorrected'), j.userFloat("caloJetMap:pt"), j.eta(),
        print "Passes loose: ", bool(pujetidFullId.product().get(i) & (1 << 2)), "medium: ", bool
        #string=""
        #string+="MVA=%.3f, " % (pujetidDiscriminant.product().get(i))
        #variables=['RMS', 'beta', 'betaClassic', 'betaStar', 'betaStarClassic', 'dR2Mean', 'dRMa
        #for var in variables:
        #  string+=var+"=%.3f, " % (getattr(pujetids.product().get(i),var)())
```

```
        #print string
        #try: genJetMatch=deltaR(j.genJet().eta(),j.genJet().phi(),j.eta(),j.phi())<0.2 and ((abs
        #except: genJetMatch=0
        #if genJetMatch:
        #    num_jets+=1.
        #    eff_loose+=bool(pujetidFullId.product().get(i) & (1 << 2))
        #    eff_medium+=bool(pujetidFullId.product().get(i) & (1 << 1))
        #    eff_tight+=bool(pujetidFullId.product().get(i) & (1 << 0))

#print num_jets
#print eff_loose/num_jets
#print eff_medium/num_jets
#print eff_tight/num_jets
```

An example how to read the re-calculated the pileup jet ID from MiniAOD in C++ is below:

Show Hide

```
edm::Handle<edm::View<pat::Jet > >; jets;
iEvent.getByLabel("selectedPatJets",jets);

Handle<ValueMap<float> >; puJetIdMVA;
iEvent.getByLabel("pileupJetIdUpdated:fullDiscriminant",puJetMva);

Handle<ValueMap<int> >; puJetIdFlag;
iEvent.getByLabel("pileupJetIdUpdated:fullId",puJetMva);

for ( unsigned int i=0; i<jets->size(); ++i ) {
      const pat::Jet & patjet = jets->at(i);
      float mva   = (*puJetIdMVA)[jets->refAt(i)];
      int    idflag = (*puJetIdFlag)[jets->refAt(i)];
      cout << "jet " << i << " pt " << patjet.pt() << " eta " << patjet.eta() << " PU JetID MVA "
      if( PileupJetIdentifier::passJetId( idflag, PileupJetIdentifier::kLoose ) {
           cout << " pass loose wp";
      }
      if( PileupJetIdentifier::passJetId( idflag, PileupJetIdentifier::kMedium ) {
           cout << " pass medium wp";
      }
      if( PileupJetIdentifier::passJetId( idflag, PileupJetIdentifier::kTight ) {
           cout << " pass tight wp";
      }
      cout << endl;
}
```

Information for 8TeV data analysis:

Show Hide
The algorithms described above can be run using the CMSSW module available here.

The list of tags to be used with different CMSSW releases is reported in the table below.

CMSSW 4X	V00-03-04	Automatically detects CMSSW version and used ultimate 4X weights when running on 4X
CMSSW 5X	V00-03-04	Used for ICHEP analyses in 5X

The package provides three classes:

- the PileupJetIdentifier class (and the *StoredPileupJetIdentifier* subset) contains all the variables used for discrimination.
- the PileupJetIdAlgo class computes all variables in *PileupJetIdentifier* and as well as the jet identification MVAs.

- the PileupJetIdProducer class is a CMSSW producer that thakes collections of *PAT* jet in input and produces *ValueMap* with *StoredPileupJetIdentifier*, the final MVAs and flags with the result of the working points evaluation.

Download the code is straightforward. The package does not have any dependency on packages not in the release.

```
cvs co -r <TAG> -d CMGTools/External UserCode /CMG/CMGTools/External
cd CMGTools/External
scram b -j 5
```

The procedure to run on PAT jets or plain reco jets is different, due to the different treatment of jet energy correction. In fact, while the input variables are independent of the jet energy corrections, the value of the final MVA depends on the corrected jet energy.

Running the producer on PAT jets

If you use PAT jets for your analysis applying the pile-up identification is as simple as running an additional sequence in you PAT process.

```
# load the PU JetID sequence
process.load("CMGTools.External.pujetidsequence_cff")

# run the PU JetID sequence
process.p = cms.Path(process.patSequence * process.puJetIdSqeuence)

# keep the PU JetID products
process.out.extend(["keep *_puJetId_*_*", # input variables
"keep *_puJetMva_*_*" # final MVAs and working point flags
])
```

This will produce the following output:

```
edmDumpEventContent patTuple.root | grep puJet

edm::ValueMap<StoredPileupJetIdentifier>    "puJetId"                  ""                 "PAT"
edm::ValueMap<float>                  "puJetMva"                "fullDiscriminant"   "PAT"
edm::ValueMap<float>                  "puJetMva"                "cutbasedDiscriminant"   "PAT"
edm::ValueMap<float>                  "puJetMva"                "simpleDiscriminant"   "PAT"
edm::ValueMap<int>                    "puJetMva"                "fullId"          "PAT"
edm::ValueMap<int>                    "puJetMva"                "cutbased"        "PAT"
edm::ValueMap<int>                    "puJetMva"                "simpleId"        "PAT"
```

For each type of algorithm (cut-based, simple and full BDTs), the producer outputs two *ValueMap*, one (**Discriminan*) with the value of the MVA and a second (**Id*) encoding the working point evaluation. This information can be accessed at analysis level in the following way:

```
edm::Handle&lt;edm::View&lt;pat::Jet&gt; &gt; jets;
iEvent.getByLabel("selectedPatJets",jets);

Handle&lt;ValueMap<float> &gt; puJetIdMVA;
iEvent.getByLabel("fullDiscriminant",puJetMva);

Handle&lt;ValueMap<int> &gt; puJetIdFlag;
iEvent.getByLabel("fullId",puJetMva);

for ( unsigned int i=0; i<jets->size(); ++i ) {
      const pat::Jet & patjet = jets-&gt;at(i);
      float mva    = (*puJetIdMVA)[jets-&gt;refAt(i)];
      int    idflag = (*puJetIdFlag)[jets-&gt;refAt(i)];
      cout &lt;&lt; "jet " &lt;&lt; i &lt;&lt; " pt " &lt;&lt; patjet.pt() &lt;&lt; " eta " &lt;&
```

```
      if( PileupJetIdentifier::passJetId( idflag, PileupJetIdentifier::kLoose ) {
           cout &lt;&lt; " pass loose wp";
      }
      if( PileupJetIdentifier::passJetId( idflag, PileupJetIdentifier::kMedium ) {
           cout &lt;&lt; " pass medium wp";
      }
      if( PileupJetIdentifier::passJetId( idflag, PileupJetIdentifier::kTight ) {
           cout &lt;&lt; " pass tight wp";
      }
      cout &lt;&lt; endl;
}
```

The *ValueMap* with *StoredPileupJetIdentifier* can be used to check the inputs to the algorithm and it allows to re-evaluate the final MVA and working points at a later stage directly on the PAT-tuples. This can be done in the following way.

```
# load the PU JetID sequence
process.load("CMGTools.External.pujetidsequence_cff")

# re-evaluate PU JetID MVA
process.p = cms.Path(process.puJetMva)

# keep the PU JetID products
process.out.extend(["keep *_puJetId_*_*", # input variables
"keep *_puJetMva_*_*" # final MVAs and working point flags
])
```

Running the algorithm on CHS jets

A specific training for CHS jets is available. In order to use it, the above instructions can be followed, replacing the producer `puJetId` with `puJetIdChs` and similarly `puJetMva` with `puJetMvaChs`.

Running the algorithm on reco jets

The producer can be run on collections of reco Jetsm however this feature has not been validated.

The final MVA has to be evaluated on corrected jets, thus the JEC have to be fed to the producer. The producer can run both on corrected and uncorrected jets, provided that the parameters are properly set. The snippet below shows how to run on a collection of uncorrected jets. To run on corrected reco jets, set the `applyJec` flag to `False` and the `inputIsCorrected` to `True`.

```
from CMGTools.External.pujetidsequence_cff import puJetId

process.recoPuJetId = puJetId.clone(
   jets = cms.InputTag("ak5PFJets"),
   applyJec = cms.bool(True),
   inputIsCorrected = cms.bool(False),
)

process.recoPuJetMva = puJetMva.clone(
   jets = cms.InputTag("ak5PFJets"),
   jetids = cms.InputTag("recoPuJetId"),
   applyJec = cms.bool(True),
   inputIsCorrected = cms.bool(False),
)

process.recoPuJetIdSqeuence(process.recoPuJetId * process.recoPuJetMva )
```

-- PasqualeMusella - 24-Apr-2012 -- KevinNash - 2017-09-27

This topic: Sandbox > TestTopic4444444
Topic revision: r2 - 2017-10-05 - EricSchanet