РЕАЛИЗАЦИЯ ЗАДАЧИ РЕГРЕССИИ В ЭКСПЕРИМЕНТЕ DSTAU

СУТЬ ЗАДАЧИ:

Цель реконструкции — извлечь кинематическую информацию о Ds из наблюдаемой топологии двойного распада.

Это важно для дальнейшего описания рождения Ds и для более точной оценки потока ν_τ в эксперименте DsTau.

В PYTHIA 8 генерируются pp-события и отбирается цепочка $D_s \to \tau\nu_\tau$.
В ROOT сохраняются truth-переменные, измеренные признаки и флаги отбора в дерево events.
по 4 топологическим переменным обучается BDT-регрессия
Качество проверяется на независимой test-выборке.

АРХИТЕКТУРА ПРОЕКТА

include/PhysicsUtils.h
src/generate_dataset.cpp
src/train_bdt.cpp
src/evaluate_model.cpp
data/processed/
datasetreg/weights/
plots/

PhysicsUtils.h

физические константы и toy-модель детектора

Generate_Dataset.cpp

генерация выборки и расчёт признаков

Train_BDT.cpp

обучение TMVA-регрессии

Evaluate_Model.cpp

применение модели и построение метрик

PHYSICSUTILS

константы и геометрия модуля

Файл объединяет все короткие функции, которые многократно используются при генерации датасета.

Сначала задаются:

- Константы масс Ds и τ, длины cτ и параметры toy-геометрии модуля.
- toy-геометрия модуля: длина модуля 4.8 mm, характерный масштаб двух слоёв 0.20 mm и массив положений эмульсионных плёнок kFilmZPositionsMm

```
namespace phys {

constexpr double kPiConst = 3.14159265358979323846;

// masses [GeV]
constexpr double kProtonMassGeV = 0.9382720813;
constexpr double kElectronMassGeV = 0.000511;
constexpr double kMuonMassGeV     = 0.105658;
constexpr double kDsMassGeV       = 1.96834;
constexpr double kTauMassGeV      = 1.77686;
constexpr double kPiMassGeV       = 0.13957;
constexpr double kRhoMassGeV      = 0.77526;
constexpr double kA1MassGeV       = 1.23000;

// c*tau [mm]
constexpr double kDsCTauMm  = 0.1499;
constexpr double kTauCTauMm = 0.08711;

// toy detector geometry / response
constexpr double kModuleLengthMm         = 4.8;
constexpr double kSensitiveLayerPitchMm  = 0.10; // kept for backward compatibility
constexpr double kTwoLayersMm            = 0.20;

// Semi-realistic film z positions inside one 4.8 mm decay module.
// replace the old uniform 0.10 mm sensitive planes with a sparse
// set of actual emulsion-film measurement planes distributed across the unit.
constexpr std::array<double, 10> kFilmZPositionsMm = {
    0.28, 0.73, 1.18, 1.63, 2.08,
    2.53, 2.98, 3.43, 3.88, 4.33
};
```

ПЕРЕВОД КИНЕМАТИКИ В ГЕОМЕТРИЮ РАСПАДА

длины пробега и углы

Функция SampleFlightLengthMm(...) генерирует длину пробега из экспоненциального закона с учётом βγ	**<L> = beta gamma c tau = (p/m) c tau длина пробега берётся из экспоненциального распределения: P(L) = (1/<L>) exp(-L/<L>)**
ProjectFlightToZMm(...) переводит длину вдоль траектории в z-проекцию внутри модуля.	Delta z = L * |p_z| / |p|
AngleMrad(...) и TrackThetaMrad(...) переводят угловую информацию в миллирадианы, чтобы дальше использовать её как измеряемые топологические признаки.	theta = arccos((a · b) / (|a| |b|))
Энергия системы центра масс для fixed-target	s = m_p^2 + m_p^2 + 2 m_p E_beam, конкретно в коде sqrt(s) = sqrt(2 m_p (E_beam + m_p))

ВИДИМОСТЬ ТРЕКОВ В ЭМУЛЬСИОННЫХ СЛОЯХ

CountSensitiveLayersCrossed(...) считает, сколько чувствительных плёнок пересекает трек на данном участке.	**N_cross = sum_i I(z_film,i > z_0 and z_film,i <= z_1)**
LocalTrackDensityProxy(...) и BasetrackEfficiency(...) задают toy-параметризацию видимости как функцию глубины, угла и плотности.	epsilon_base = Clamp(0.992 - 0.015 d - 0.010 Clamp(theta_track/400,0,1) - 0.012 rho_loc , 0.92 , 0.997)
SampleDetectedLayers(...) делает Монте-Карло-выбор: из числа пересечённых слоёв выбирается, сколько реально детектировано.	N_det ~ Binomial(N_cross , epsilon_base)

```
104  inline int CountSensitiveLayersCrossed(const double zStartMm,
105                                         const double zFlightMm,
106                                         const double moduleLengthMm = kModuleLengthMm,
107                                         const double /*layerPitchMm*/ = kSensitiveLayerPit
108      if (zFlightMm <= 0.0) return 0;
109      const double z0 = Clamp(zStartMm, 0.0, moduleLengthMm);
110      const double z1 = Clamp(zStartMm + zFlightMm, 0.0, moduleLengthMm);
111      if (z1 <= z0) return 0;
112
113      int crossed = 0;
114      for (std::size_t i = 0; i < kFilmZPositionsMm.size(); ++i) {
115          const double zFilm = FilmZAt(i, moduleLengthMm);
116          if (zFilm > z0 && zFilm <= z1 + 1e-12) ++crossed;
117      }
118      return crossed;
119  }
120
121  inline int EstimateDetectedFilmsFromArm(const double armLengthMm,
122                                          const double moduleLengthMm = kModuleLengthMm) {
123      if (armLengthMm <= 0.0) return 0;
124      const double meanPitch = std::max(MeanFilmPitchMm(moduleLengthMm), 1e-6);
125      const int estimate = static_cast<int>(std::floor(armLengthMm / meanPitch)) + 1;
126      return std::max(1, std::min(static_cast<int>(kFilmZPositionsMm.size()), estimate));
127  }
128
129  inline double LocalTrackDensityProxy(const double depthFrac,
130                                       const double trackThetaMrad,
131                                       const int nCrossed) {
132      const double depthTerm = Clamp(depthFrac, 0.0, 1.0);
133      const double angleTerm = Clamp(trackThetaMrad / 350.0, 0.0, 1.0);
134      const double occupancyTerm = Clamp(static_cast<double>(nCrossed) /
135                                         static_cast<double>(kFilmZPositionsMm.size()),
136                                         0.0, 1.0);
137      return Clamp(0.55 * depthTerm + 0.20 * angleTerm + 0.25 * occupancyTerm, 0.0, 1.0);
138  }
```

МОДЕЛИРОВАНИЕ ИЗМЕРЕНИЯ

EmulsionFlightSigmaMm(...) задаёт ошибку измерения длины пробега.	**sigma_L = sigma_L0 * (1 + 0.10 / max(L,0.20)) * (1 + 0.25 d)**
MeasureFlightLengthEmulsion(...) превращает истинную длину в измеренную: добавляет гауссов шум и квантование.	L_meas = max(0 , Quantize(L_true + N(0,sigma_L) , 0.001 mm))
MeasureAngleFast(...) и MeasureAnglePrecision(...) строят уже измеренные углы для двух стадий отбора.	theta_fast^meas = max(0 , theta_true + N(0,sigma_fast))- fast theta_prec^meas = max(0 , theta_true + N(0,sigma_prec))- presicion

```
171  inline double EmulsionFlightSigmaMm(const double flightMm,
172                                      const double depthFrac,
173                                      const double baseSigmaMm = 0.00020) {
174      const double lenFactor = 1.0 + 0.10 / std::max(flightMm, 0.20);
175      const double depthFactor = 1.0 + 0.25 * Clamp(depthFrac, 0.0, 1.0);
176      return baseSigmaMm * lenFactor * depthFactor;
177  }
178
179  inline double MeasureFlightLengthEmulsion(const double trueLengthMm,
180                                            const double depthFrac,
181                                            const double baseSigmaMm,
182                                            TRandom3& rng) {
183      const double sigma = EmulsionFlightSigmaMm(trueLengthMm, depthFrac, baseSigmaMm);
184      const double smeared = std::max(0.0, trueLengthMm + rng.Gaus(0.0, sigma));
185      return std::max(0.0, Quantize(smeared, 0.001));
186  }
188   inline double FilmCountResolutionFactor(const int nDetectedFilms) {
189       if (nDetectedFilms <= 2) return 1.00;
```

FilmCountResolutionFactor(...) Чем больше плёнок реально увидели, тем лучше angular resolution	**f_film(N) = 1.00, N <= 2 0.90, N = 3 0.82, N = 4 max(0.68, sqrt(2/N)), N >= 5**
Fast angular resolution	sigma_fast = sigma_fast,0= 2.5 mrad * sqrt(L_ref / max(L_arm,L_ref)) * (1 + 0.15 d) * (1 + 0.05 Clamp(theta_track/200,0,2)) * f_film(N_det)
Precision angular resolution	sigma_prec = max(0.14 , sigma_prec,0 * sqrt(L_ref / max(L_arm,L_ref)) * (1 + 0.10 d) * (1 + 0.03 Clamp(theta_track/200,0,2)) * f_film(N_det))

GENERATE_DATASET

- Файл выполняет две задачи: находит в PYTHIA нужную физическую цепочку распада и строит detector-aware датасет, пригодный для машинного обучения.

- На выходе создаётся ROOT-файл ds_signal.root с деревьями events и truth, а также набор диагностических гистограмм, которые позволяют контролировать качество генерации и отбора.

event generation → signal selection → measured features → ROOT trees

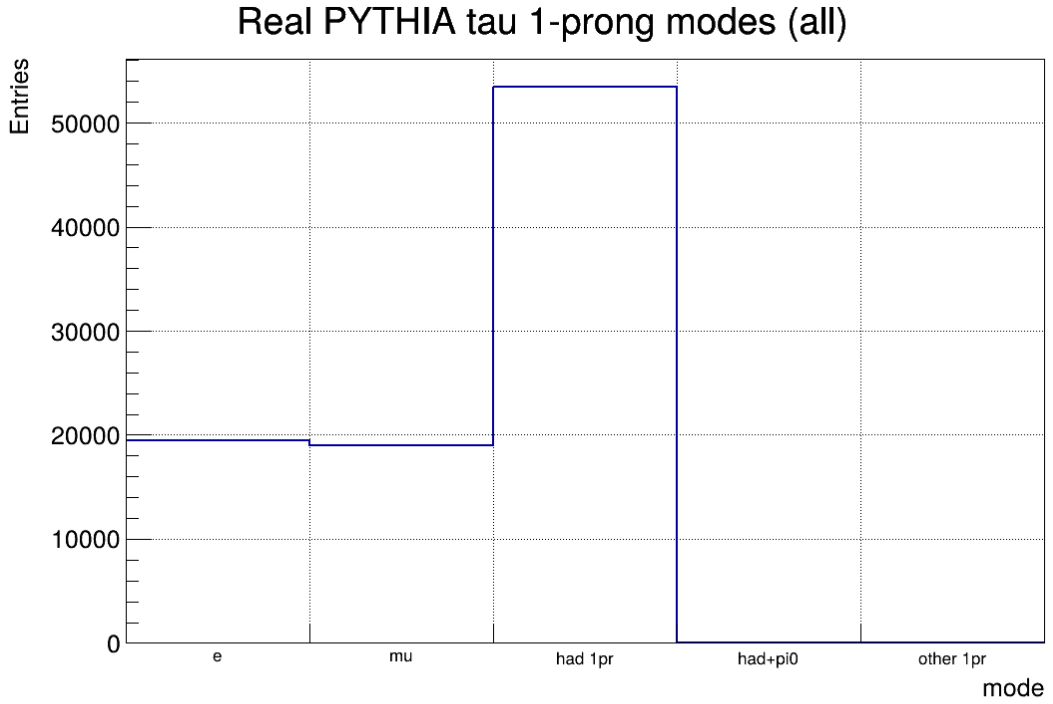
КАК ОПРЕДЕЛЯЕТСЯ СИГНАЛЬНАЯ ТОПОЛОГИЯ

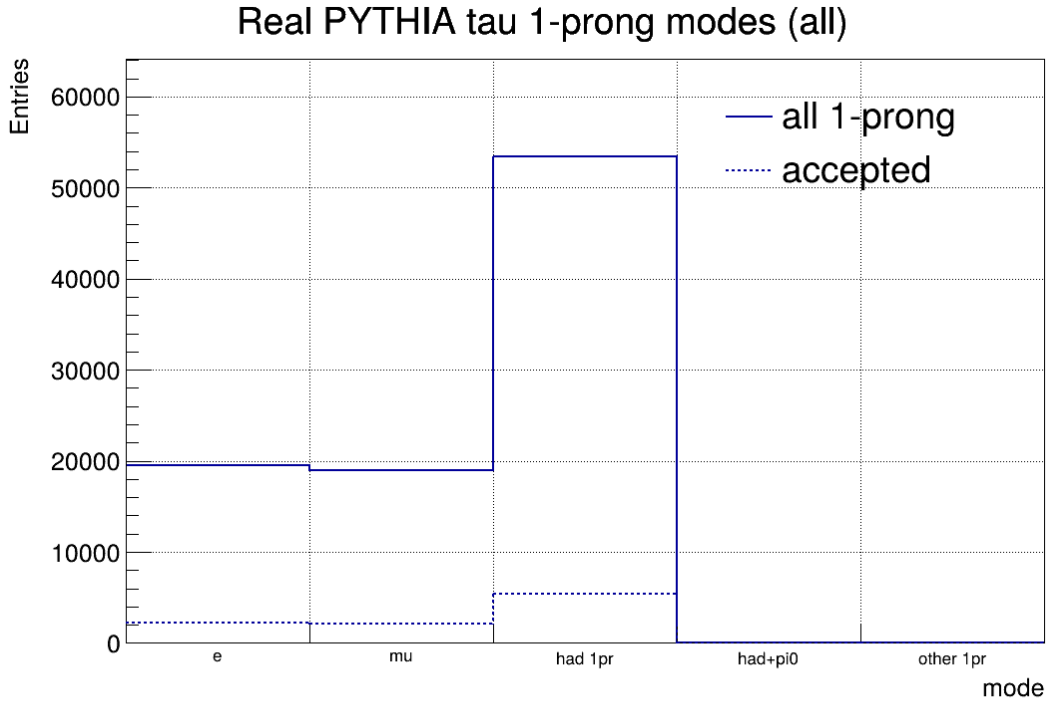
Отбор распада D_s → τν_τ, τ → 1-prong

Сначала из event record выбираются только частицы D_s.
Затем проверяется, что у них есть прямой распад D_s → τν_τ.
После этого анализируется распад τ, и остаются только события типа 1-prong, где есть ровно одна заряженная конечная частица.

```
71  bool FindDirectDsToTauDecay(const Pythia8::Event& event,
72                              const int iDs,
73                              int& iTau,
74                              int& iNuTau) {
75      iTau = -1;
76      iNuTau = -1;
78      const auto daughters = event[iDs].daughterList();
79      for (const int idx : daughters) {
80          if (idx <= 0 || idx >= event.size()) continue;
81          const int absId = std::abs(event[idx].id());
82          if (absId == 15 && iTau < 0) {
83              iTau = idx;
84          } else if (absId == 16 && iNuTau < 0) {
85              iNuTau = idx;
86          }
87      }
88
89      return (iTau >= 0 && iNuTau >= 0);
90  }
```

```
92  TauDecayInfo BuildTauDecayInfo(const Pythia8::Event& event, const int iTau) {
93      TauDecayInfo info;
94
95      if (iTau <= 0 || iTau >= event.size()) return info;
96
97      const auto descendants = event[iTau].daughterListRecursive();
98      for (const int idx : descendants) {
99          if (idx <= 0 || idx >= event.size()) continue;
100
101         const auto& d = event[idx];
102         if (!d.isFinal()) continue;
103
104         if (IsNeutrinoPdg(d.id())) {
105             ++info.nNeutrinos;
106             continue;
107         }
108
109         TLorentzVector p4(d.px(), d.py(), d.pz(), d.e());
110         info.visibleP4CM += p4;
111         ++info.nVisibleFinal;
112
113         if (d.isCharged()) {
114             ++info.nChargedFinal;
115             if (info.chargedProngPdg == 0) info.chargedProngPdg = d.id();
116         } else {
117             ++info.nNeutralVisibleFinal;
118             if (std::abs(d.id()) == 111) info.hasPi0 = 1;
119         }
```


Real PYTHIA tau 1-prong modes (all)
Entries
50000
40000
30000
20000
10000
0
e
mu
had 1pr
had+pi0
other 1pr
mode

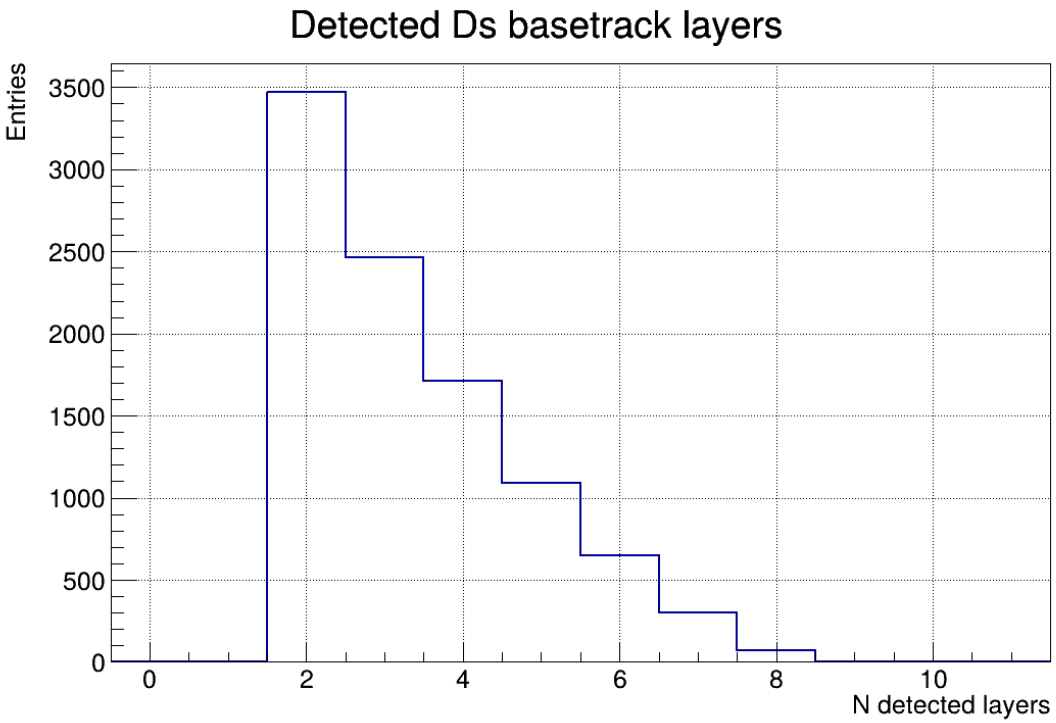

Real PYTHIA tau 1-prong modes (all)
Entries
60000
50000
40000
30000
20000
10000
0
all 1-prong
accepted
e
mu
had 1pr
had+pi0
other 1pr
mode

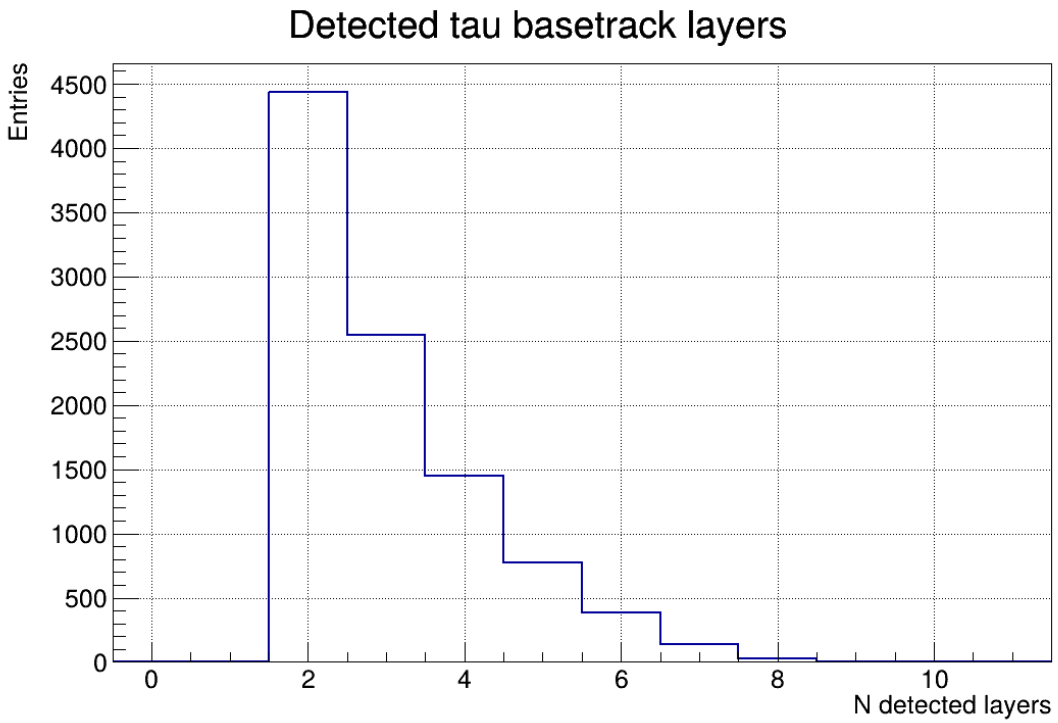
ПОСТРОЕНИЕ ТОПОЛОГИЧЕСКИХ ПЕРЕМЕННЫХ

После выбора сигнального события код вычисляет:
truth-длины пробега fl_ds_true, fl_tau_true,
truth-углы kink_ds_tau_true, kink_tau_x_true.

Затем на основе toy-модели детектора строятся уже измеренные величины:
fl_ds, fl_tau, kink_ds_tau, kink_tau_x.
Именно они подаются на вход регрессионной модели.

```
460         fl_ds_true = static_cast<float>(phys::SampleFlightLengthMm(pDsLab, phys::kDsCTauMm, rng));
461         fl_tau_true = static_cast<float>(phys::SampleFlightLengthMm(tauLab, phys::kTauCTauMm, rng));
462         kink_ds_tau_true = static_cast<float>(phys::AngleMrad(pDsLab.Vect(), tauLab.Vect()));
463         kink_tau_x_true = static_cast<float>(phys::AngleMrad(tauLab.Vect(), visLab.Vect()));
464
465         const double dzDsTrue = phys::ProjectFlightToZMm(pDsLab, fl_ds_true);
466         const double dzTauTrue = phys::ProjectFlightToZMm(tauLab, fl_tau_true);
467         const double zTau = z_vertex + dzDsTrue;
468         const double zEnd = zTau + dzTauTrue;
469
470         const double dsDepthFrac = phys::Clamp(static_cast<double>(z_vertex) / opt.moduleLengthMm, 0.0, 1.0);
471         const double tauDepthFrac = phys::Clamp(zTau / opt.moduleLengthMm, 0.0, 1.0);
```


Detected Ds basetrack layers
Entries
3500
3000
2500
2000
1500
1000
500
0
0
2
4
6
8
10
N detected layers


Detected tau basetrack layers
Entries
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
2
4
6
8
10
N detected layers

FAST STAGE И PRECISION STAGE

```
cand.pass_fast_stage =
    (cand.pass_ds_visible &&
     cand.pass_tau_visible &&
     cand.fl_ds < opt.maxFlightMm &&
     cand.fl_tau < opt.maxFlightMm &&
     cand.kink_tau_x_fast >= opt.minKinkTauXMrad) ? 1 : 0;

if (cand.pass_fast_stage) {
    hCutflow.Fill(6);

    const double armDsTau = std::max(
        opt.minTwoLayersMm,
        std::min(static_cast<double>(cand.fl_ds_true), static_cast<double>(cand.fl_tau_true))
    );

    cand.kink_ds_tau = static_cast<float>(phys::MeasureAnglePrecision(
        cand.kink_ds_tau_true,
        armDsTau,
        tauDepthFrac,
        phys::TrackThetaMrad(tauLab),
        opt.sigmaAnglePrecisionMrad,
        rng
    ));
```

```
cand.pass_precision_stage =
    (cand.pass_fast_stage &&
     cand.kink_ds_tau >= opt.minKinkDsTauMrad &&
     cand.kink_tau_x >= opt.minKinkTauXMrad) ? 1 : 0;

if (cand.pass_precision_stage) hCutflow.Fill(7);

cand.accepted_evt = cand.pass_precision_stage ? 1 : 0;
if (cand.accepted_evt) hCutflow.Fill(8);

truth.Fill();

if (!cand.accepted_evt) continue;

cand.split = (rng.Uniform() < 0.8) ? 0 : 1;
events.Fill();

hTauModesAcc.Fill(cand.tau_mode);
hDsLayersDetected.Fill(cand.ds_layers_detected);
hTauLayersDetected.Fill(cand.tau_layers_detected);

++acceptedCount;
```

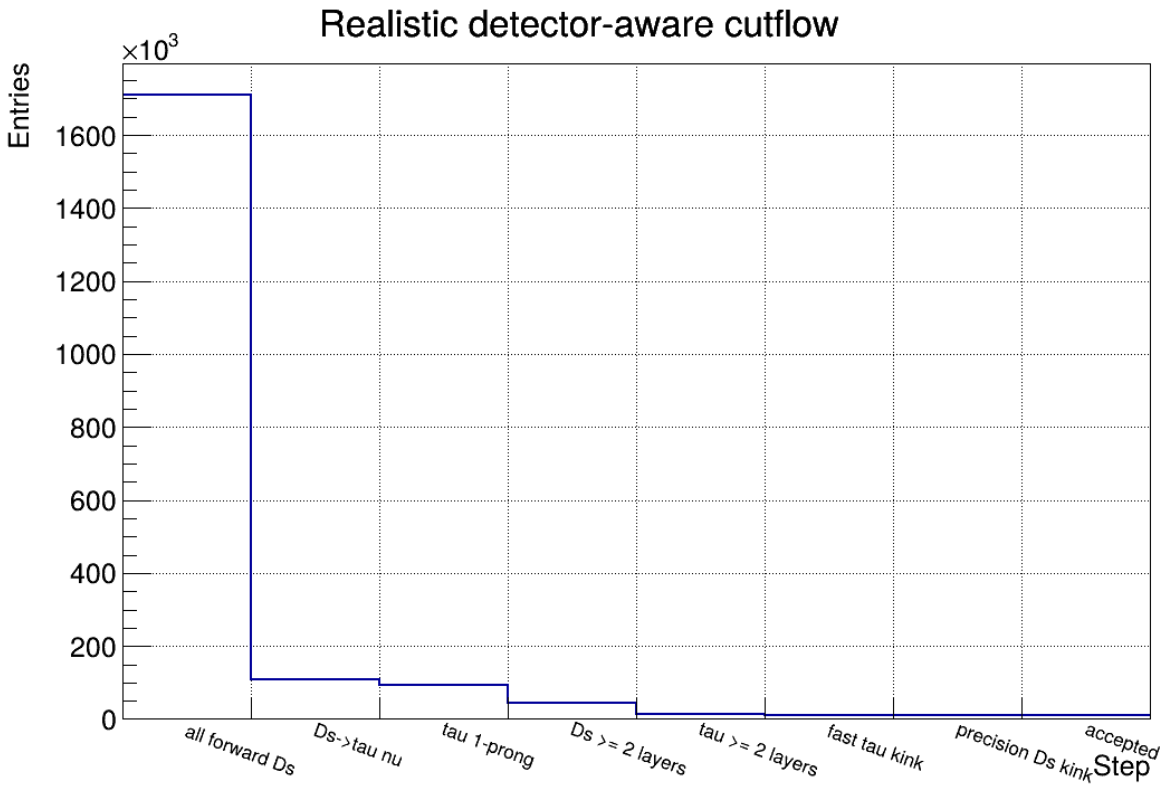
После построения признаков код выполняет двухступенчатый отбор.

На fast stage проверяется:

видимость обеих вершин;
разумность длин пробега;
достаточность грубого τ-kink.

На precision stage выполняется точный пересчёт углов и накладываются финальные пороги на Ds-kink и τ-kink.
После precision stage событие считается достаточно измеримым для включения в итоговое дерево events

СТРУКТУРА ВЫХОДНОГО ДАТАСЕТА



Файл сохраняет два дерева:

truth — хранит все forward Ds-кандидаты и флаги прохождения отдельных стадий отбора.

events — содержит только итоговые accepted-события, которые реально используются для обучения и теста.

Резкое уменьшение статистики после геометрических и угловых требований объясняется тем, что сигнал должен быть измеримым внутри короткого модуля с достаточно выраженными kink-углами.

TRAIN_BDT

чтение дерева и построение train/test выборок

- Файл открывает ds_signal.root и берёт дерево events.
  Далее по полю split строятся две подвыборки: split==0 — train, split==1 — test.
- Если в дереве есть флаг in_reco_sample, он дополнительно используется как фильтр.

```
std::string trainCut = "split==0";
std::string testCut  = "split==1";
if (hasRecoFlag) {
    trainCut = "in_reco_sample==1 && " + trainCut;
    testCut  = "in_reco_sample==1 && " + testCut;
}
```

```
TTree* trainTree = tree->CopyTree(trainCut.c_str());
TTree* testTree  = tree->CopyTree(testCut.c_str());
```

КАКИЕ ПЕРЕМЕННЫЕ ИДУТ В TMVA

В TMVA::DataLoader добавляются ровно те четыре переменные, которые были заложены в постановке задачи:

fl_ds
fl_tau
kink_ds_tau
kink_tau_x

Целевая переменная — `p_ds_true`.
Дополнительные величины вроде `tau_mode`, `xF`, `eta_ds` идут как spectators и нужны для диагностики, а не для обучения

```
loader->AddVariable("fl_ds",       "FL_{Ds}",                          "mm",   'F');
loader->AddVariable("fl_tau",      "FL_{#tau}",                        "mm",   'F');
loader->AddVariable("kink_ds_tau", "#delta#theta_{Ds#rightarrow#tau}", "mrad", 'F');
loader->AddVariable("kink_tau_x",  "#delta#theta_{#tau#rightarrowX}",   "mrad", 'F');

if (hasAcceptedEvt) {
    loader->AddSpectator("accepted_evt", "accepted_evt", "", 'I');
} else if (hasAccepted) {
    loader->AddSpectator("accepted", "accepted", "", 'I');
}
if (hasTauMode) {
    loader->AddSpectator("tau_mode", "tau_mode", "", 'I');
}
if (hasDsLayers) {
    loader->AddSpectator("ds_layers_detected", "ds_layers_detected", "", 'I');
}
if (hasTauLayers) {
    loader->AddSpectator("tau_layers_detected", "tau_layers_detected", "", 'I');
}
if (hasXF) {
    loader->AddSpectator("xF", "xF", "", 'F');
}
if (hasEta) {
    loader->AddSpectator("eta_ds", "eta_ds", "", 'F');
}
```

НАСТРОЙКА BDTG И ЗАПУСК ОБУЧЕНИЯ

используется метод `BDTG` — gradient boosted decision trees.
BDTG выбран как компактная нелинейная модель для небольшого числа физических признаков; , хорошо работает на небольшом числе входных переменных и умеет ловить нелинейные зависимости.

В коде явно задаются основные гиперпараметры:
`NTrees=400`, `Shrinkage=0.05`, `BaggedSampleFraction=0.70`, `MaxDepth=4`.
После этого запускается стандартный цикл:
`TrainAllMethods()`, `TestAllMethods()`, `EvaluateAllMethods()`.

```
factory.BookMethod(
    loader.get(),
    TMVA::Types::kBDT,
    "BDTG",
    "!H:!V:"
    "NTrees=300:"
    "BoostType=Grad:"
    "Shrinkage=0.05:"
    "UseBaggedBoost:"
    "BaggedSampleFraction=0.60:"
    "nCuts=40:"
    "MaxDepth=3:"
    "MinNodeSize=2.5%"
);

factory.TrainAllMethods();
factory.TestAllMethods();
factory.EvaluateAllMethods();
```

EVALUATE_MODEL

Чтение весов и применение Reader

- evaluate_model.cpp загружает веса TMVARegression_BDTG.weights.xml, снова открывает дерево events и создаёт TMVA::Reader.
- В Reader добавляются те же четыре входные переменные, что использовались при обучении.
- Дополнительно в Reader повторно объявляются spectators, чтобы схема данных совпадала с той, что использовалась в DataLoader.

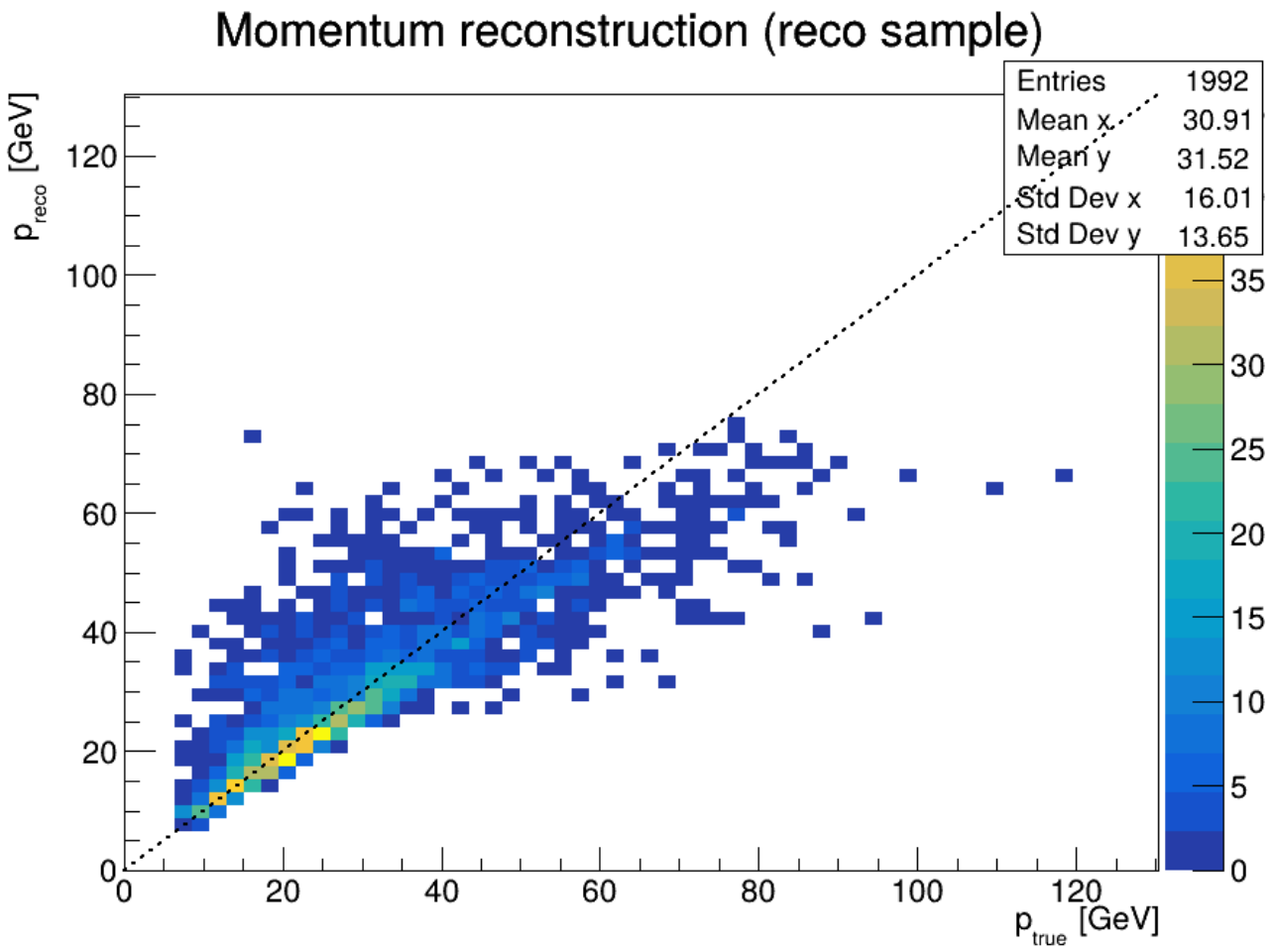
РАСЧЁТ ОШИБОК И МЕТРИК

На этапе оценки используются только тестовые события: `split == 1`.
Для каждого события вычисляется:

гистограммы relative_error,
карты pred_vs_true,
расчёта MAE, RMSE,
построения bias vs p_true,
построения resolution vs p_true.

MOMENTUM RECONSTRUCTION: P_RECO VS P_TRUE

- Основная плотность событий остаётся вдоль диагонали, значит модель действительно извлекает устойчивую кинематическую корреляцию из распадной топологии.
- При больших импульсах видно расширение облака и сжатие отклика ниже диагонали. Это указывает на ухудшение чувствительности топологических признаков в высокоимпульсной области.



RELATIVE MOMENTUM ERROR & BIAS AND RESOLUTION VS TRUE MOMENTUM

- Gaussian fit по центральной части распределения используется как оценка core resolution; хвосты при этом не игнорируются физически, а отделяются от центрального режима как более сложные случаи реконструкции.
- Рост resolution с p_true означает, что разброс ошибки увеличивается в жёсткой кинематике.
  Смена знака bias показывает систематическую компрессию шкалы: внизу диапазона модель слегка завышает оценку, а вверху — занижает.