Department of Physics and Astronomy
Ruprecht-Karls-Universität Heidelberg

Bachelor thesis
in Physics

submitted by

Xuan-Xuyen Nguyen
born in Darmstadt

2021

Identification of Monte-Carlo photon conversions from simulations in ALICE using XGBoost

This Bachelor Thesis has been carried out by

Xuan-Xuyen Nguyen

at the

Institute of Physics in Heidelberg

under the supervision of

Prof. Dr. Klaus Reygers

Zusammenfassung

Diese Arbeit befasst sich mit der Klassifizierung primärer Photonen aus sogenannten V^0-Kandidaten mithilfe des XGBoost-Algorithmus. Die Kandidaten stammen aus einer Monte-Carlo Simulation in ALICE. Dabei erkennt man Photonen durch das Finden von Photon-Konversionen unter den Kandidaten. Diese müssen von Untergrund-Daten unterschieden werden, welche sich aus Zerfällen und zufälligen kombinatorischen Kandidaten zusammensetzt. Letzteres sind Teilchenpaare, welche nicht vom selben Mutterteilchen stammen. Zudem werden die V^0-Kandidaten durch eine Vorselektion mehr auf die Photon-Konversionen beschränkt. Durch Monte-Carlo simulierte Kollisionen ist die wahre Identität der V^0-Kandidaten bekannt. Bei den hoch entwickelten Simulationen werden kollidierende Teilchen erzeugt und darauffolgende Interaktionen mit den ALICE-Detektoren simuliert. In diesem Fall handelt es sich um Simulationen von Proton-Proton und Blei-Blei-Kollisionen. Während die Proton-Proton-Kollisionen als Referenz dienen, sind die Blei-Blei-Kollisionen interessant, weil sich Effekte des Quark-Gluonen-Plasma besonders deutlich zeigen. Die Kollisionen werden unter anderem vom V^0-finder ausgewertet, welcher anschließend V^0-Kandidaten vorschlägt. In dieser Arbeit werden Modelle mit den unterschiedlichen Algorithmen auf den simulierten Daten für binäre Klassifizierung trainiert und mit der normalerweise verwendeten Schwellenwert-Klassifizierung verglichen. Neben dem XGBoost-Algorithmus wird auch der Random-Forest-Algorithmus verwendet. Eine Hyperparameter-Optimierung wird für die XGBoost Modelle durchgeführt, um die mögliche Verbesserung zu erforschen. Im interessanten Fall des Blei-Blei-Kollisionssystems können die trainierten Modelle im Vergleich zum Schwellenwert-Modell bei gleicher Signaleffizienz Photonen mit etwa 30% höherer Reinheit identifizieren. Dabei sind die Unterschiede zwischen den trainierten Modellen vernachlässigbar klein.

Abstract

This thesis deals with the classification of primary photons from so-called V^0-candidates using the XGBoost algorithm. These candidates originate from Monte-Carlo Simulations in ALICE. The algorithm identifies photons by finding photon conversions among the candidates. It is necessary to distinguish the photon conversions from the background data, which comprises decays and random combinatorial candidates. The latter describes particle pairs, which do not originate from the same mother particle. Moreover, a preprocessing cut restricts the set of V^0-candidates more to the photon conversions. The use of Monte-Carlo simulated collisions makes it possible to know the true identity of the so-called V^0-candidates. The sophisticated simulations create colliding particles and simulate subsequent interactions with the ALICE-detectors. In this case, proton-proton and lead-lead collisions are simulated. While proton-proton collisions serve as a reference, the lead-lead collisions are interesting because the effects of the quark-gluon plasma are more apparent. Then the V^0-finder evaluates these collisions and subsequently proposes the V^0-candidates. This thesis deals with the training of machine-learning models with simulated data for binary classification and the comparison to the typically used cut-based model. The main task is to train models with two machine-learning algorithms on simulated data for binary classification and compare them with the typically used cut-based model. Besides the XGBoost algorithm, the comparison also includes the random-forest algorithm. The hyperparameter optimization is part of the comparison. It should explore the possible improvement of an XGBoost model. In contrast to the cut-based model, the XGBoost models achieve around 30% higher purity at the same signal efficiency in the lead-lead collision system. At the same time, the difference between the machine-learning models is negligible.

Contents

1 Introduction **1**
1.1 Photons 1
1.2 Heavy-Ion-Collisions 1
1.3 A Large Ion Collider Experiment 2
1.4 Research objectives 5

2 Classification **7**
2.1 Decision Trees 9
2.2 Boosting 9
2.3 Gradient Boosting 9
2.4 Extreme Gradient Boosting 10

3 Using XGBoost for photon classification **13**
3.1 The Monte-Carlo simulated dataset 14
3.1.1 V0-identification and V0-candidates 14
3.1.2 Features 17
3.2 Hyperparameter search and Training 18
3.3 Feature importance 21
3.4 Evaluation of XGBoost models 32
3.4.1 Comparison with cut-based model 33
3.4.2 Performance of XGBoost models at high purities 39
3.4.3 Comparison with default lead-lead model 43
3.4.4 Comparison with random forest 44
3.4.5 Performance of the random forest model at high purities 48

4 Conclusion and discussion **51**

A Appendix **53**

Chapter 1

Introduction

This chapter introduces the reader to the necessary terms for understanding the physical part of photon identification. The study of today's experimental particle physics extends way beyond the study of particle collisions. Results in particle physics also teach about the development of the universe. Nowadays, researchers expect that the universe fulfilled the conditions for the existence of the quark-gluon-plasma (QGP) 10 μs after the big bang [1]. Thus, research on the QGP also gives an understanding of the density and temperature of the early stages of our universe. Calculations of quantum chromodynamics (QCD) predict this state of matter. It is a state where hadrons devolve into a plasma of quarks and gluons. It only occurs when the energy density is high enough, such as in heavy-ion-collisions or the early state of our universe [2]. Researchers also expect quark-gluon plasma to exist in neutron stars [3]. The current technology enables heavy-ion-collisions at high energy densities making research on quantum chromodynamics (QCD) possible. The Large-Hadron-Collider (LHC) at CERN in Geneva enables such high energetic heavy-ion-collisions. Located inside the ring at one of the interaction points is the A Large Ion Collider Experiment (ALICE) for QCD research at high multiplicities. Although the QGP is not directly observable, there are different ways to detect its properties. One way to measure the properties of the QGP is to study the photons, which originate from the QGP. Therefore, the main task here is to train a classifier to identify primary photons from V^0-tracks, which get their name by their shape. V^0-particles are neutral and include particles, which decay or convert into two daughter particles forming the V-shaped track.

1.1 Photons

High energy photons, such as these originating from the QGP, convert into an electron and an oppositely charged positron. This process is known as photon conversion or pair formation. Thus, the classification task is to find photon conversions among the V^0-candidates. Photons are essential probes for the QGP because they are produced during the complete collision evolution and do not interact with the medium. We group the photons into direct and decay photons based on their origin. As the name already suggests, decay photons originate from hadronic decays. They represent the predominant proportion of photon yield. Therefore, direct photons are the remaining photons, which do not originate from decays [2]. These direct photons originate from the QGP.

1.2 Heavy-Ion-Collisions

Under normal circumstances, quarks and gluons occur in a confined state. Therefore, they typically do not occur isolated. When heavy ions collide, they go through different states.

Figure 1.1 depicts the collision evolution. The first state is the pre-equilibrium, where the partons scatter soft and hard. Hard interactions are interactions with a large momentum transfer. They lead to hard partons and jets. Soft interactions include those, where such jets do not occur. At high energy density, quarks and gluons are deconfined. This state is called the quark-gluon-plasma. In other words, the partons reach thermal equilibrium through interaction, as the quarks and gluons merge into a quark-gluon soup. Eventually, the QGP will cool down due to the expansion, which causes the energy density to drop. The QGP will turn into hadronic matter. The final state of the collision evolution is known as freeze-out. Here, hadrons are free and do not interact with each other anymore [2].

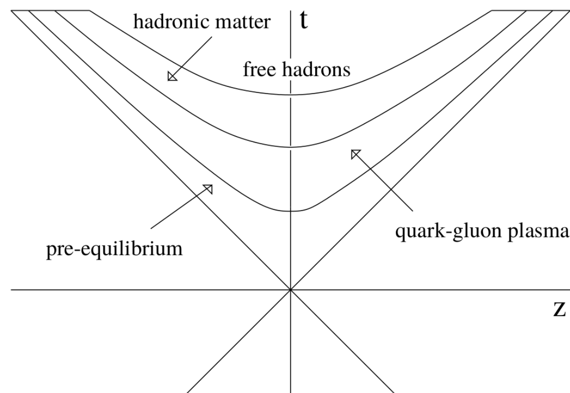
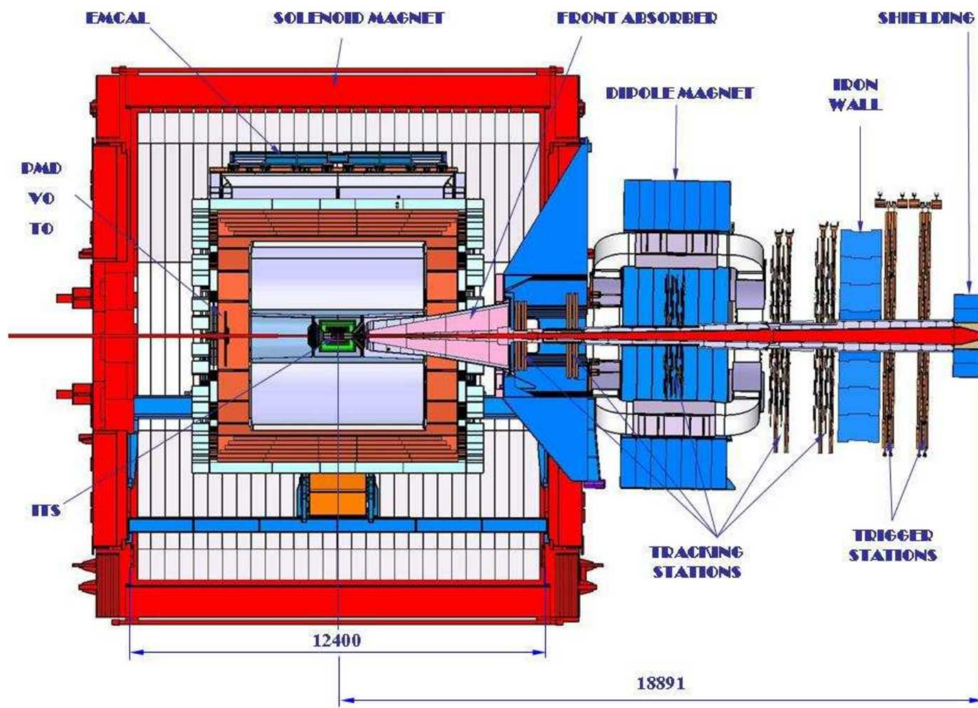


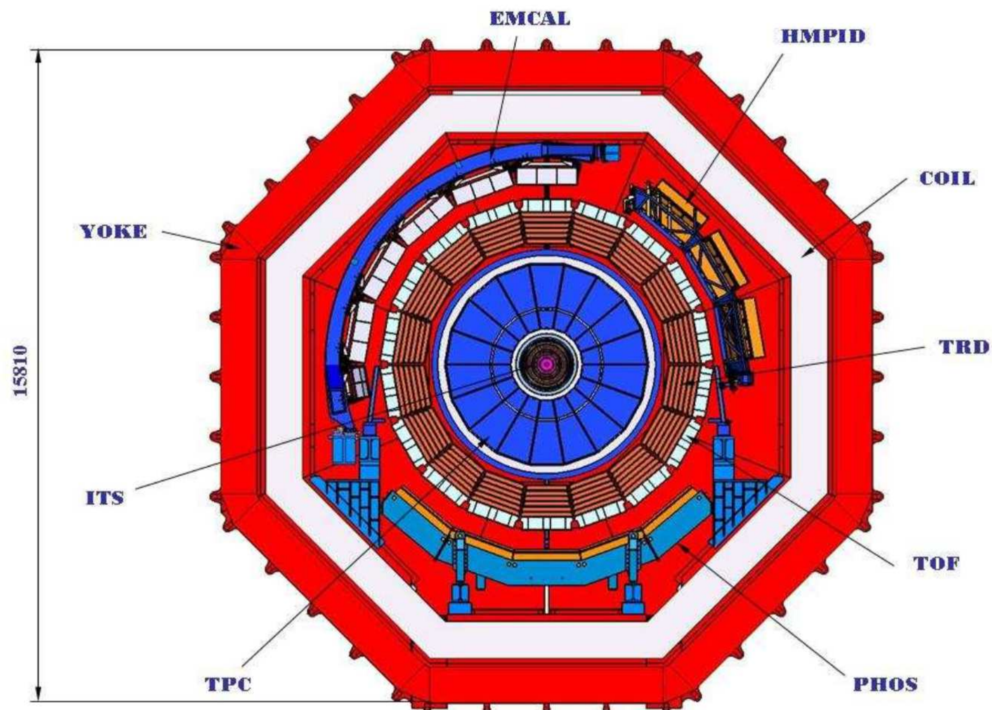
Figure 1.1: Collision evolution [4].

1.3 A Large Ion Collider Experiment

ALICE is one of the main experiments at the 27 km ring accelerator LHC. It has multiple sub-detectors and provides excellent particle identification at high multiplicities.
Figure 1.2a depicts the detectors from a side view, where one can distinguish the muon arm on the right side and the central barrel on the left. The detectors of the central barrel, which are relevant for this thesis, are the Inner Tracking System (ITS), the Time Projection Chamber (TPC), and the Time-of-Flight (TOF) detector. Figure 1.2b illustrates the detectors from a front perspective.
Since 2019, there is a Long Shutdown (LS2), which includes various detector updates. However, the Monte-Carlo simulated data in this thesis does not incorporate these updates as they reflect the situation before the shutdown.
The task of the **ITS** is particle tracking in the proximity of the particle collision and the determination of the primary vertex with high precision (better than 100μm). Together with the TPC it identifies secondary particles with low momentum (below 200 MeV/c) using the energy loss dE/dx. Thus, it provides data for primary and secondary vertex reconstruction. The primary vertex describes the position of the particle collision. Primary particles emerge from this vertex. Analogously, the secondary vertex is the location at which the primary particle decays or converts into the secondary particles. The upgrades of the LS2 led to an improvement of the tracking performance and the momentum resolution of the ITS. Previously it consisted of six cylindrical silicon detector layers surrounding the beam pipe at radii between 39 mm and 430 mm. If a charged particle crosses the detector, it ionizes the material and thus creates measurable electrons and electron holes [5]. After the LS2 upgrade, the ITS consists of seven silicon pixel detector layers of the so-called ALPIDE chips. This development of the ALICE collaboration is a combination of the Monolithic Active Pixel Sensor (MAPS) with the readout-circuit [6, p. 85].



(a) Side view of the ALICE detector.



(b) Front view of the ALICE detector.

Figure 1.2: The ALICE detector [5, p. 5].

A more precise measurement of the energy loss happens in the **TPC**, the main detector of ALICE in the central barrel. It serves as particle identification and tracking system. It was optimized to measure the momentum of particles and the energy loss $\mathrm{d}E/\mathrm{d}x$ with high precision at very high charged particle densities in central lead-lead collisions. Inside the TPC, charged particles pass through the gas chamber and ionize the gas. A uniform electrostatic drift field then guides the electrons to the two read-out planes on the end caps. The TPC is also used for vertex determination [5, p. 54]. Before LS2, the TPC had to operate in a gated mode to suppress space charge-induced drift field distortions. In this mode, the gating grid

closes the read-out chambers by default and only opens for one drift time interval triggered by the L1 trigger. The updated TPC does not require gating because of the replacement of the Multi-Wire Proportional Chamber (MWPC) based read-out chambers by Gas Electron Multiplier (GEM) detectors [6, p. 85].

The **TRD** tracks and identifies particles. It also creates a fast trigger for charged particles with high momentum. This detector is good at distinguishing pions from electrons. Electrons with a high momentum above 1 GeV/c are particularly well distinguishable. As the name suggests, it measures transition radiation. Highly relativistic charged particles emit this radiation when they cross thin material layers of the TRD, which have different dielectric constants. The radiation only extends to the X-ray domain for highly relativistic particles with a Lorentz factor $\gamma \gtrsim 1000$. Therefore, lighter and faster particles, such as electrons or positrons, can be easily distinguished from the other particles, such as pions. The detector has 18 super-modules. Each super module consists of 30 read-out modules allowing the TRD to host 540 read-out chambers in total. But due to a not-installed stack, only 522 read-out chambers were available [7].
As the name already indicates, the **TOF** detector measures the time of flight of charged particles. It is a gas detector with Multi-gap Resistive-Plate Chamber (MRPC) technology. The MRPC creates a strong and uniform magnetic field inside the gaseous volume of the detector. When charged particles cross the ionization gas, they trigger an avalanche process creating a measurable signal [5, p. 74].
For the data regarding a particle's position, a coordinate system is necessary. The two relevant systems here are the Cartesian and the polar coordinate system. Figure 1.3 shows the coordinate systems. The origin lies in the interaction point (IP). In the Cartesian system, the parameters x, y and z describe the position of a particle. Aligned with the beam axis is the z-axis, which faces away from the myon arm. The y-axis points upwards and the x-axis points left. The parameters θ, ϕ and r describe the position of a particle in a polar system. The inclination θ is the angle between the z-axis and the vector r, which starts at the origin and ends at the particle's position. The azimuthal angle describes the angle between the x-axis and the projection of r to the xy-plane. From θ, one can calculate the pseudo-rapidity η:

$$\eta = -\ln\left[\tan\left(\theta/2\right)\right]$$

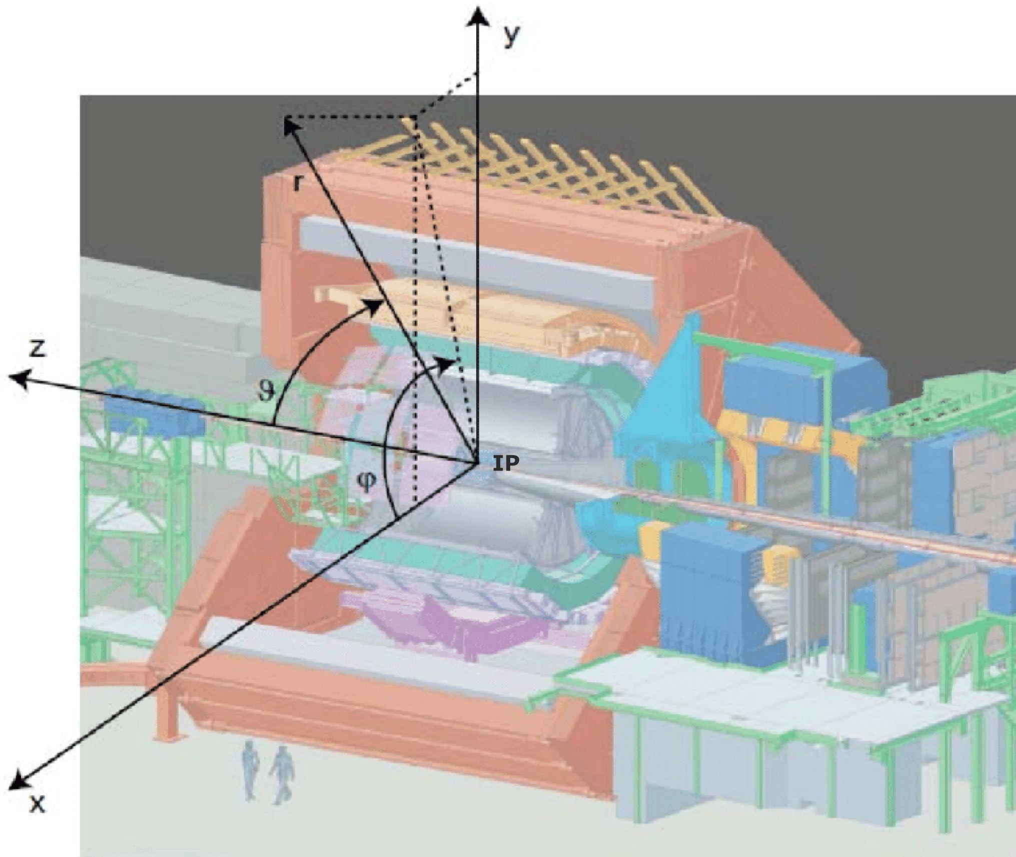


Figure 1.3: ALICE coordinate system [8].

1.4 Research objectives

The main research objective of this thesis is to identify photon conversions among Monte-Carlo simulated V^0-candidates using machine-learning algorithms. Chapter 3 documents the training of sophisticated machine learning models, namely XGBoost models. The goal is to classify V^0-candidates and find electron-positron pairs from primary photon conversions. In this process, hyperparameter optimization should lead to hyperparameters, which result in better classifiers. The training and the evaluation of the models happen on different datasets to ensure that the examination is not affected by overfitting. The same chapter shows the comparison between the XGBoost models and the cut-based model. Here, the focus is on the signal efficiency at different transverse momenta. For this purpose, an algorithm adjusts the purity of the models. As one would use these XGBoost models to create photon datasets of high purity, the XGBoost models also have to complete classification tasks with high purity. It is also of interest how a simpler machine learning model performs this task. Therefore, the random forest model trains on the same training data and completes the corresponding comparisons as the XGBoost models. In the end, this thesis should show how well the machine learning algorithms perform this classification task in comparison to the cut-based algorithm.

Chapter 2

Classification

This chapter introduces relevant machine learning terms. It starts with purity and efficiency, which will be used repeatedly in later chapters. Step-by-step, it will then introduce the reader to the XGBoost algorithm.
Classification is the task in which one needs to assign a data instance to a group based on its properties. In this thesis, the data instance is a V^0 candidate, which needs to be identified either as a primary photon conversion or as any other V^0 candidate, the so-called "background". To be precise, a classifier f gives a response $Y \in \{0,1,\dots,C-1\}^N$ to an input X, where C is the number of classes and N is the number of signal instances. Here, one performs binary classification ($C = 2$) by discriminating direct photons against the background. The input has the dimension $N \times D$, where D is the number of features. The features of the given Monte-Carlo simulated datasets comprise the energy loss, the transverse momentum, and other measurement data from the ALICE detectors. The classifier assigns a class to each input:

$$f(X) = Y \tag{2.1}$$

There is always a modeling error and the prediction Y is not always the same as the reality $\hat{Y}$. Because the classification task is binary, one can call $Y = 0$ "negative" (signal is not a primary photon) and $Y = 1$ "positive" (signal is a primary photon). The confusion matrix is therefore simple:

		Prediction		
		p	**n**	**total**
Ground truth	**p***	True Positive	False Negative	P*
	n*	False Positive	True Negative	N*
	total	P	N	

Table 2.1: Confusion matrix for a binary classifier.

In the following, some abbreviations will be used:

- TP for True Positive
- TN for True Negative
- FP for False Positive
- FN for False Negative

The error rate is defined as

$$\text{error} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} , \tag{2.2}$$

which is just the proportion of falsely classified instances among all classifications.
If one instead divides the sum of correctly classified instances by the number of all instances, the result is the accuracy:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = 1 - \text{error}. \tag{2.3}$$

Two important evaluation parameters for this thesis are the purity and the signal efficiency.
The purity is the proportion of correctly classified direct photons divided by the number of instances classified as direct photons:

$$\text{purity} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.4}$$

It is also known as precision or the **P**ositive **P**redictive **V**alue (PPV).
A commonly used terminology is the signal efficiency, which describes how many direct photons the classifier has identified among all the direct photons in the data set:

$$\text{efficiency} = \frac{\text{TP}}{\text{TP} + \text{FN}} . \tag{2.5}$$

The signal efficiency is also known as recall, sensitivity, hit rate or **T**rue **P**ositive **R**ate (TPR).
In particle physics, it is common to use the terms purity and signal efficiency, while in the area of Machine Learning the terms precision and recall are more common.
One can group classifiers into discriminative and generative models. Discriminative models learn the probability of the class y given the feature input x: $\hat{p}\left(y|x\right)$.
The nearest neighbor classifier and decision trees belong to the discriminative models.
Generative models make use of Bayes formula:

$$p\left(Y|X\right) = \frac{p\left(X|Y\right) \cdot p\left(Y\right)}{p\left(X\right)} \tag{2.6}$$

Where $p\left(Y|X\right)$ is the posterior, $p\left(X|Y\right)$ is the likelihood, $p\left(Y\right)$ is the prior and $p\left(X\right)$ is the evidence or feature density.
These models learn the likelihood, the prior, and the feature density. The naive Bayes classifier is a generative model.

2.1 Decision Trees

Decision trees are widely used base learners for ensembles. For instance, the random forest or XGBoost can use decision trees as base learners. The decision tree can approximate discrete functions. The resulting tree structure is flow-chart-like and splits at each of its internal nodes. Each split is a decision. A branch of the tree denotes the classification rules, and each leaf of the decision tree represents a class. A decision tree can be more transparent concerning the decision process. In contrast, the decision of black-box models, such as neural networks, is usually not comprehensible.

2.2 Boosting

Boosting is an ensemble method. It is the combination of sequentially created weak learners such as decision trees. The idea is to yield a precise model by combining many weak classifiers, where the term "weak" refers to poor prediction accuracy. The algorithm starts with a classifier, which trains on the training set. In the subsequent iterations, the algorithm calculates the prediction errors. It also computes the new instance weights emphasizing the falsely classified instances. A new classifier then trains on the weighted training set to learn the complicated cases and joins the ensemble. Tree boosting uses decision trees as base classifiers and is a popular and successful machine learning method [9]. In this thesis, the XGBoost algorithm uses tree boosting. But by creating such an ensemble, the decision rules of boosted decision trees become less comprehensible. Thus, boosting algorithms, such as boosted decision trees, gradient boosting, and XGBoost, are black-box algorithms.

2.3 Gradient Boosting

But in contrast to other tree boosting algorithms, the gradient boosting algorithm trains new trees on the pseudo-residual hence the name gradient boosting. In a nutshell, the term gradient in the name comes from the pseudo-residual, which is the negative gradient of the loss function. Given a training set $\{(x_i,y_i)\}_{i=1}^{n}$, a differentiable loss function $\mathcal{L}(y,F(x))$, and the number of desired trees T, the algorithm starts with a constant model, which minimizes the loss function.

$$F(x) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(y_i,c) \tag{2.7}$$

Next, it builds up trees sequentially with each new tree correcting the old model until the model reaches the desired number of trees. In each boosting step $t = 1...T$, the algorithm computes the pseudo-residual for each data instance i, which, roughly speaking, is the difference between prediction and truth.

$$r_{it} = -\frac{\partial \mathcal{L}(y_i,F_{t-1}(x)))}{\partial F_{t-1}(x)} \quad ,i = 1,...,N \tag{2.8}$$

The pseudo-residual is the negative derivative of the loss function. It is called pseudo-residual because of its similarity with the residual. In case the loss function is the mean squared error, the pseudo-residual is equal to the residual. In a simple form, where y^* is the ground truth, and $F(x)$ is the prediction of the model, the mean squared error is

$$\frac{1}{2}(y^* - F(x))^2. \tag{2.9}$$

The pseudo-residual is $y^* - F(x)$, which is also the residual. Because generally, the negative derivative of a loss function is not the residual, it is called pseudo-residual. After the computation of pseudo-residuals, a new tree $R_t(x)$ learns the pseudo-residuals. The updated model is

the sum of the old model and the pseudo-residual tree multiplied with a coefficient.

$$F_t(x) = F_{t-1}(x) + \gamma_t R_t(x) \tag{2.10}$$

Hereafter is the exact algorithm, which was described above:
Algorithm: [see 10]

1. Start with a constant model (which minimizes the loss function)

$$F(x) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, c) \tag{2.11}$$

   Here, c is the predicted value for all instances.

2. Build up trees sequentially with each new tree correcting the old model until desired number of trees is reached.
   For t=1 to T:

   (a) Compute pseudo-residuals for each instance:

$$r_{it} = -\left[\frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{t-1}(x)} , i = 1,...,N \tag{2.12}$$

   This is the gradient, which gives the name.
   It is called pseudo-residual, because of it is similarity to the residual. If one sets the mean squared error as the loss function, the pseudo-residual is also the residual:

$$\text{mse-loss} = \frac{1}{2}(y^* - F(x))^2 \tag{2.13}$$

   (simplified)

$$\frac{\partial L}{\partial F} = y^* - F(x) \tag{2.14}$$

   (residual)

   (b) Now, a tree, let's call it $R_m(x)$, will learn the pseudo-residuals.

   (c) The Lagrangian multiplier γ will be computed by minimizing the loss of the updated model:

$$\gamma_m = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma R_m(x_i)) \tag{2.15}$$

   (d) The updated model is:

$$F_m(x) = F_{m-1}(x) + \gamma_m R_m(x) \tag{2.16}$$

2.4 Extreme Gradient Boosting

The open-source library XGBoost is the acronym for e**X**treme **G**radient **Boost**ing. This machine-learning algorithm was initially released in March 2014 by Tianqi Chen and Carlos Guestrin and quickly gained popularity due to its success on Kaggle, a platform for machine learning competitions and education [11].
In 2015, for instance, 17 out of 29 winning solutions used XGBoost. One of the categories won with XGBoost concerned event classification in high energy physics. That is mentioned in the paper "XGBoost: A Scalable Tree Boosting System", which also shows that this algorithm is

independent of the input scale. Therefore the input does not have to be standardized [12].
What makes XGBoost "extreme" is that it enables L1 and L2 regularization to prevent overfitting. Moreover, it implements parallel processing on multiple threads to make training and prediction faster. The tree-pruning capability enables a non-greedy training of the trees. The downside of the advantages is the vast number of tunable hyperparameters. That means hyperparameter search is more complicated. Although XGBoost is a black-box algorithm, it provides a table of feature importance. Thus, it enables the user to understand the decision rules better. Section 3.3 describes the different metrics to measure feature importance.

Chapter 3

Using XGBoost for photon classification

This chapter starts with a motivation for the usage of the XGBoost algorithm. Then, the reader learns about the Monte-Carlo simulated dataset of V^0-candidates. As described in the previous chapter, the data is separated into a training and validation subset. The hyperparameter optimization uses a subset of the training data to optimize the training. After the hyperparameter optimization, the XGBoost models are trained with optimized and default hyperparameters. Later, the random forest model is trained with the same training dataset. All models are first evaluated at the same purity as the cut-based model. Then the machine learning models also predict at high purities like mentioned before. These evaluations happen on the validation dataset, such that the results are not prone to overfitting.

The cut-based approach is the currently used method to distinguish photons from background. That means that the decisions rely on manually found feature thresholds. The code in the appendix (A) describes the decision process clearly. As it shows, the prediction relies on three conditions. First, there is a restriction on the kinetic range. Then the mother particle has to fulfill various requirements. At last, the daughter particles have to meet diverse prerequisites. A detailed description of the decision rule can be found in chapter 5 of the Ph.D. thesis *Measurement of neutral mesons and direct photons in Pb-Pb collisions at* $\sqrt{s_{NN}} = 2.76$ *TeV with the ALICE experiment at the LHC* [2]. Because there is a great variety of software libraries and algorithms available, the question arises if there are alternatives to make this process faster, better and easier. The idea to use boosted decision trees and XGBoost is not new to the LHC. For instance, with the ROOT toolkit TMVA, various studies at CMS and ATLAS used boosted decision trees [13]. But as mentioned in section 2.4, the XGBoost algorithm is also well established in physics. The main argument is that with XGBoost non-trivial relationships among the features can be learned. Therefore, XGBoost became interesting for this thesis. The adjustment to different collision systems is more efficient because feature thresholds do not have to be found by hand. That means creating data models could be easier, faster, and better. Because it also does not depend on distance metrics, it is not cursed by dimensionality. That means that one can create useful data models with a relatively small amount of data.

Models for which the similarity relies on distance metrics would need an unfeasible amount of data with each new dimension. The nearest neighbor classification relies on distance metrics. One would need enough training data in the neighborhood of the new data point to classify it, which is not always possible. It is particularly complicated with high-dimensional data. In contrast to that, XGBoost, and generally speaking, tree boosting methods, start with a constant model. That means there is only one neighborhood. By boosting, the neighborhoods form and shrink to the "real regions" [14]. Although there are many advantages, one should keep the disadvantages in mind. It is trivial that the model can only be as good as the training data. The given datasets are Monte-Carlo simulations and might not represent the true data

perfectly.

3.1 The Monte-Carlo simulated dataset

The dataset from the experiments does not contain the true identity of particles. Therefore, it is beneficial to use the Monte-Carlo simulation because it gives information on a particle's identity. In this thesis, two collision systems were relevant. The datasets contain simulations of proton-proton and the lead-lead collisions. As mentioned before, the proton-proton collisions serve as a reference. Lead-lead collisions are interesting because they reach high energy densities over a large volume such that the effects of the QGP are more apparent. However, the simulation does not include the QGP. The Monte-Carlo simulation starts with a so-called event generation, which creates the simulated lead or proton. That includes their mutual interaction as well as the momentum distribution. Here, the Heavy Ion Jet Interaction Generator (HIJING) was used to simulate lead-lead collisions. The event generator PYTHIA was used to produce simulations of proton-proton collisions. The second step is the detector response, which includes the particle interaction with the detector material and the corresponding detector response. The resulting impact on the particle and its trajectory is also a part of the simulation. In this thesis, the GEANT (Geometry And Tracking) simulates the detector response [2]. The simulated dataset contains labels for each V^0-candidate, which the next section will describe.

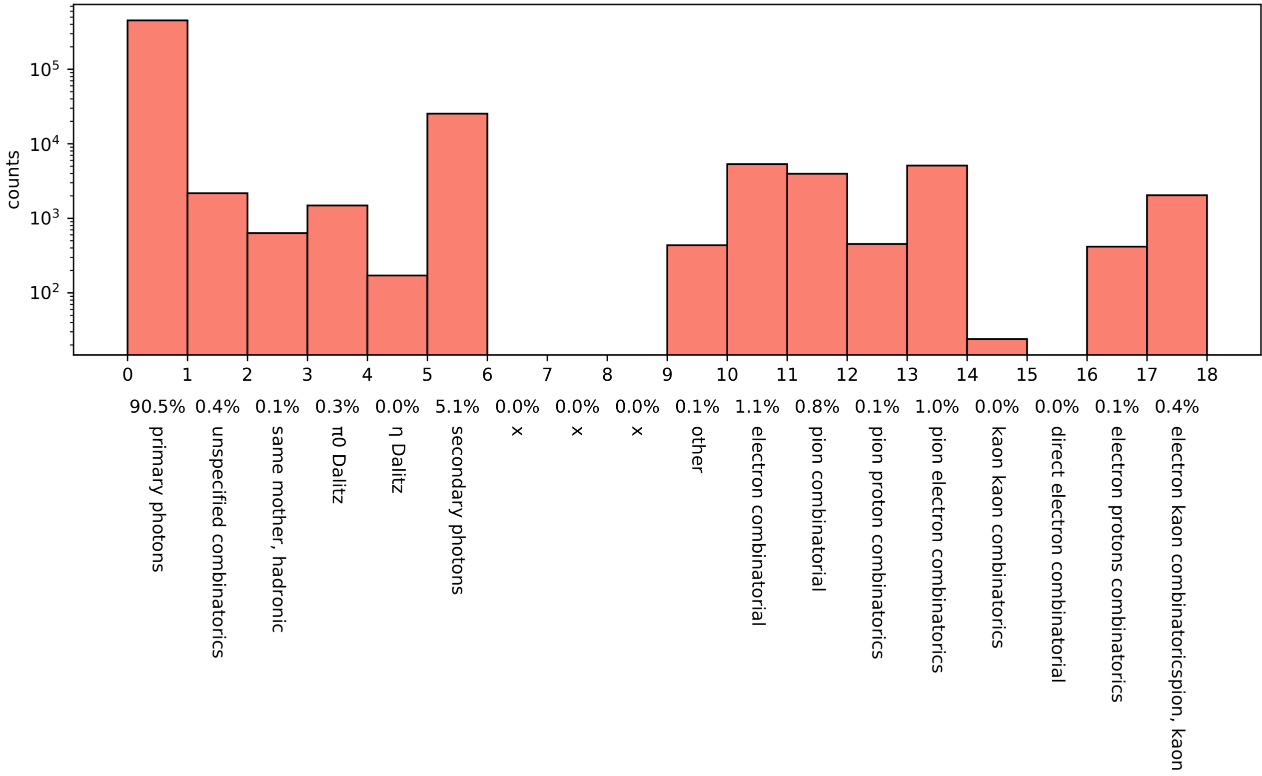
3.1.1 V^0-identification and V^0-candidates

Each instance in a dataset is a V^0 candidate proposed by the V^0-finder, which analyses V-shaped trajectories. Their attributes are the different detector responses, such as the energy loss or the goodness of tracking-fit. V^0 candidates include V^0 particles and random particles, which happen to have a V-shaped trajectory. V^0 particles get their name by the distinct V-shaped track their daughter particles leave.

The figures 3.1a and 3.1b show the different kinds of V^0-candidates for photon conversion. The figures also show the number of data instances per V^0-candidate for each dataset of the two collision systems. As already described in the beginning, the V^0-candidates comprise the photon conversions and the background. Photons can be identified by the pair production, which leaves the distinct V-track. The other candidates are background data, which include particle decays, secondary photon conversions, and random combinatorial candidates. There were preprocessing steps, which already rejected some background. The following section will describe these V^0-candidates. As can be seen in the figures, the first bar represents the abundance of primary photons, which are the focus of the classification in this thesis. These have to be distinguished from other candidates. In this thesis, primary photons are defined as photons, which originate from the primary vertex. Thus, it comprises photons from π^0 and η decays near the primary vertex. The neutral pion (π^0) most probably decays into two photons ($\sim 98.8\%$) but occasionally decays into an electron, a positron, and a photon ($\sim 1.2\%$) [15]. The latter is known as a Dalitz decay.

$$\pi^0 \rightarrow 2\gamma \qquad (\sim 98.8\%)$$
$$\pi^0 \rightarrow e^+ + e^- + \gamma \qquad (\sim 1.2\%)$$

(a) proton-proton dataset.



(b) lead-lead dataset.

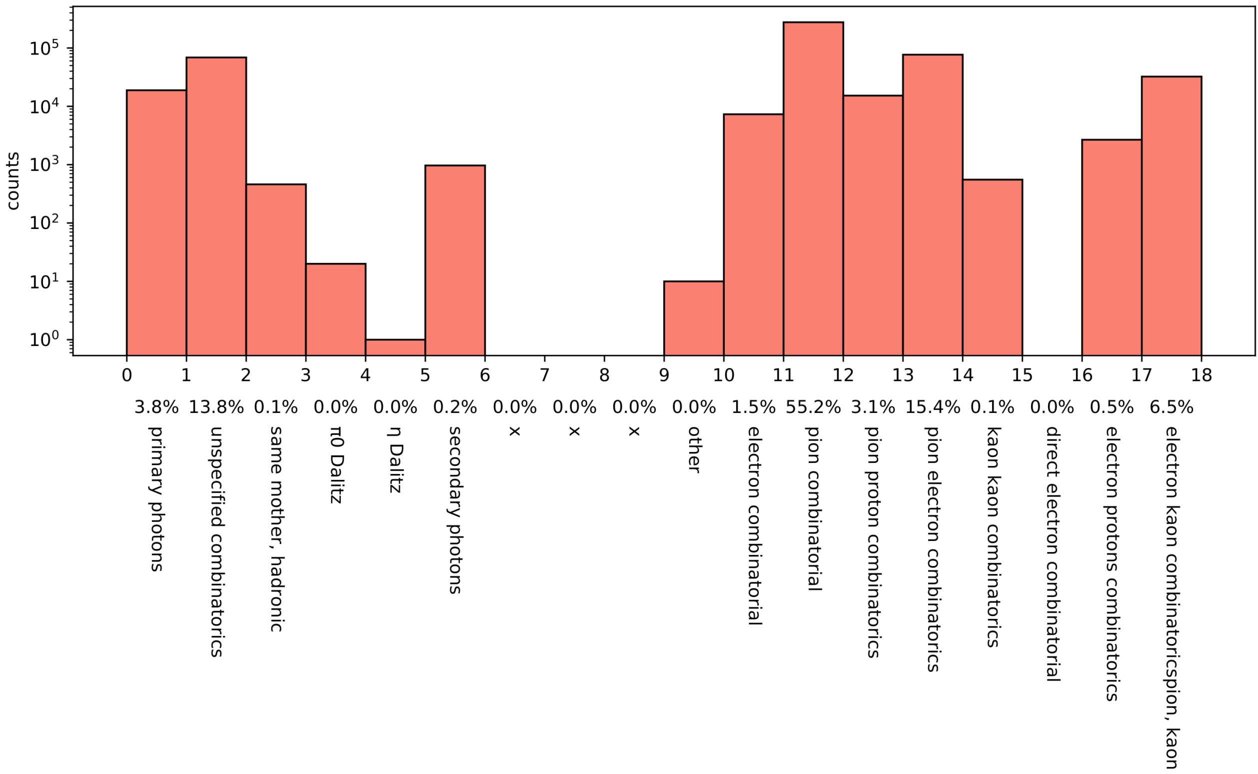


Figure 3.1: Abundance of V^0-candidates.

The neutral η-meson usually decays into three pions or two photons. The pions can be charged or neutral.

$$\eta \to 2\gamma \qquad (\sim 39\%)$$
$$\eta \to 3\pi^0 \qquad (\sim 33\%)$$
$$\eta \to \pi^+ + \pi^- + \pi^0 \qquad (\sim 23\%)$$
$$\eta \to \pi^+ + \pi^- + \gamma \qquad (\sim 4\%)$$

Photons from QGP would also count as primary photons, but the simulation does not include the QGP. Primary photons are the target of the classification task in this thesis.
As figure 3.1a shows, the proportion of primary photons in the proton-proton dataset is rather high, as it comprises 90.5% of the V^0-candidates. The secondary photons are the largest contribution to the background with 5.1%. With a proportion of 1.1%, the electron-electron and pion-electron combinations contribution is not insignificant. In the lead-lead dataset, the primary photon proportion is lower (3.8%). There is significantly more background. Here, the pion-pion combinations account for 55.2% of the total V^0-candidates. The pion-electron combinations amount to 15.4%. Unspecified combinations account for 13.8%. These are unspecified pairs, which also happen to have a V-shaped trajectory. Secondary photons usually emerge from the collision cascade further away from the primary vertex.
The hadronic kind is a V^0-decay into hadronic daughter particles. It is, for instance, the decay of massive, unstable subatomic particles like the Λ^0. The Λ^0 decays into a nucleon and a pion. In this process, the decay into a proton and a π^- is most probable. The oppositely charged daughter particles travel in a V-shaped trajectory.

$$\Lambda^0 \to p + \pi^- \qquad (\sim 64\%)$$
$$\Lambda^0 \to n + \pi^0 \qquad (\sim 36\%)$$

The Dalitz decay is the decay of mesons into two leptons and a photon, like the neutral pion decay or the eta decay, as is shown above. The two oppositely charged leptons will create a V-shaped trajectory. The combinatorics mentioned in the figure are random charged particles, which created a V-shaped trajectory by chance. There are many different charged particle combinations for this case. Electron-electron, pion-pion, pion-proton, pion-electron, kaon-kaon, direct electrons, electron-proton, and electron-kaon are combinations, where each pair represents an own type of V^0-candidate. The dotted trajectory in figure 3.2 could be a trajectory of these combinations. Figure 3.2 illustrates the decay of a V^0 particle. The illustration also shows the pointing angle Θ, which is a feature of the dataset.

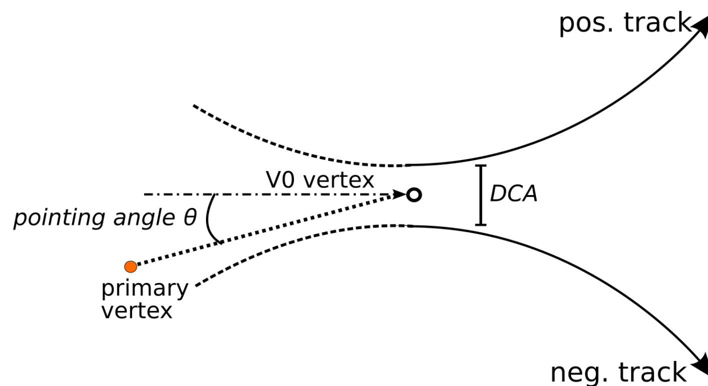


Figure 3.2: Decay of a V^0 particle [16].

3.1.2 Features

ptPositron	Transversal momentum of a positron.
thetaPositron	θ is used to calculate the pseudo-rapidity of the positron candidate. It is shown in figure 1.3.
dEdxPositronTPC	The energy loss of the positron measured at the TPC.
tofPositron	Time of flight measured for a positron.
nSigmaTOFPositron	The deviation from the electron band in units of the standard deviation. Particle identification with the help of the particle velocity β vs momentum p plot of the TOF.
fracClsTPCPositron	Fraction of clusters in the TPC that measured positrons.
clsITSPositron	Number of clusters triggered by positrons in the ITS.
dEdxPositronITS	Energy loss dE/dx of the positron measured with the ITS.
clsTPCPositron	Number of clusters triggered by positrons in the TPC.
nSigmaITSPositron	The deviation from the electron band in units of the standard deviation. Particle identification with the help of the energy loss dE/dx vs momentum p plot of the ITS.
ptElectron	Transversal momentum of an electron.
thetaElectron	θ is used to calculate the pseudo-rapidity of the electron candidate.
dEdxElectronTPC	Energy loss dE/dx of the electron measured with the TPC.
nSigmaTPCElectron	The deviation from the electron band in units of the standard deviation. Particle identification with the help of the energy loss dE/dx vs momentum p plot of the TPC.
tofElectron	Time of flight of an electron.
nSigmaTOFElectron	The deviation from the electron band in units of the standard deviation. Particle identification with the help of the particle velocity β vs momentum p plot of the TOF.
fracClsTPCElectron	Fraction of clusters in the TPC that measured electrons.
clsITSElectron	Number of clusters triggered by electrons in the ITS.
dEdxElectronITS	Energy loss dE/dx of the electron measured with the ITS.
clsTPCElectron	Number of clusters triggered by electrons in the TPC.
nSigmaITSElectron	The deviation from the electron band in units of the standard deviation. Particle identification with the help of the energy loss dE/dx vs momentum p plot of the ITS.
photonQt	The momentum transfer in transverse direction.
photonAlpha	Longitudinal momentum asymmetry between the secondary tracks (in case of photon conversion: positron and electron): $\frac{p_L^+ - p_L^-}{p_L^+ + p_L^-}$
photonPsiPair	The angle between the plane spanned by the decay products and the plane orthogonal to the magnetic field. (See figure 3.3)
photonCosPoint	$\cos(\Theta_p)$. Where Θ_p is the angle between two vectors: 1. The vector from the primary vertex to the position of the photon conversion. 2. The momentum vector of the photon at the time of decay.
photonInvMass	The invariant mass of the photon candidate.
photonX, photonY, photonZ	Cartesian coordinates.
photonPhi	The angle in the spherical coordinate.
photonR	The distance from the primary vertex at which the photon converts into an electron and a positron. The distance in the cylindrical coordinates.

pt	Transverse momentum.
theta	The inclination as shown in figure 1.3
chi2ndf	The chi-squared test of the tracking.
kind	The identity of the V^0-candidate.

Table 3.1: The table lists all the features of the datasets and describes each attribute name.

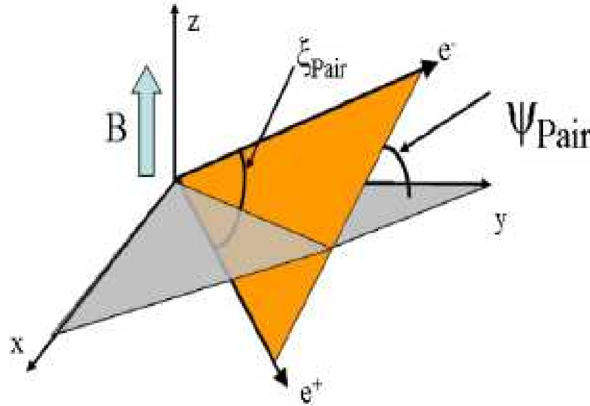


Figure 3.3: Ψ_{pair} shown for a photon conversion into electron and positron [17].

3.2 Hyperparameter search and Training

This section documents the hyperparameter search and the training. Both processes happen on the training dataset. The decision was taken that the models should be independent of the transverse momentum p_T. The idea is that a classifier should perform well independent of the transverse momentum. For instance, the model should not exploit the fact that there are only a few photons at high transverse momentum and reject all candidates with high momentum. Instead, it should perform well constantly across the whole transverse momentum range. During training and validation, this feature is therefore hidden. Early tests have shown that the transverse momentum does not contribute much to the model performance. Thus, the missing feature has no significant impact on the model performance, which coincides with the observation. The term hyperparameter describes the parameters, which affect the model's training. Hyperparameter optimization is the process of finding the best parameters for a model, such that the loss minimal. Table 3.2 shows the tested hyperparameters.

Parameter	**explanation**
max_depth	The maximum depth of a tree. A higher value leads to more complex models but increases the risk for overfitting[1].
min_child_weight	Minimal weight of a child. If the weight of the child is lower than this threshold, the child will become a leaf. That means one sets the minimal purity of a leaf.
gamma	The minimum loss at which the leaf node will be split (default: 0; If the loss is greater than 0, the algorithm creates a new partition and the leaf becomes a node.)
subsample	The fraction of random instances from the dataset that is used in each iteration. It has the value 1 by default. So, the model is trained with all instances by default. But a lower number could prevent overfitting.
colsample_bytree	The fraction of random features used in each training iteration.
reg_alpha/reg_lambda	L1 and L2 regularization. The technique that uses L1 regularization is also known as Lasso (Least Absolute Shrinkage and Selection Operator) Regression and adds the product of a scalar α and the absolute value of the weight of the features to the loss function, which one has to minimize. The L2 regularization technique is known as the ridge regression, which adds the product of a scalar λ and the squared feature weights to the loss function.

Table 3.2: Hyperparametersearch on the proton-proton dataset.

Attached to this thesis, one can find the corresponding code in the appendix A and A. The model will output probabilities. The output is 0 if the model is sure that the instance is not a photon. As one would expect, with a higher predicted probability, the model is more confident that the candidate is a primary photon. If the model is sure that the V^0-candidate is a photon, then the output would be 1. The learning objective is *binary:logistic* for the hyperparameter optimization and the training. Thus, the binary classification uses logistic regression. The loss function for this learning objective is shown in equation 3.1.

$$\mathcal{L} = -\frac{1}{N} \cdot \left(-y \log\left(y^*\right) + (y-1)\log\left(1-y^*\right)\right) \tag{3.1}$$

There are different algorithms for the search of hyperparameters. Scikit-learn has a method for hyperparameter optimization and cross-validation called *GridSearchCV*. This method implements the grid search algorithm to search for the optimal hyperparameters from a given parameter set. It also uses cross-validation to check the generalization performance of the model. Models will be trained with every combination of the given hyperparameters. In each training iteration, the k-fold cross-validation algorithm splits the training set into k equally large subsets. Then, the model is trained on $k-1$ subsets and completes an evaluation on the remaining subset. This method gives more precise measurements of the model's performance because it is less prone to statistical variation of the dataset. It decreases the risk that the model is just lucky with the validation set and undeservedly gets a great prediction score. Different evaluation metrics are available to measure the performance of the classifiers. One of the evaluation metrics is the so-called accuracy. The problem is that it strongly depends on the data set. Therefore, there are different evaluation metrics, such as the *AUCPR*, the *AUCROC* and the average precision score. The **A**rea **U**nder the **C**urve for the **P**recision-**R**ecall curve or the **R**eceiver-**O**perating-**C**haracteristic gives us a good summary of the model performance and is more reliable than the accuracy. An ideal model would reach 100% purity independent of the recall. It would then have an AUCPR and an average precision score of 1. Note that purity is also known as precision. Moreover, the signal efficiency is also known as recall. The

average precision score results from the following formula:

$$\text{avg.prec.} = \sum_i \left(\text{signal efficiency}_i - \text{signal efficiency}_{i-1}\right) \cdot \text{purity}_i \tag{3.2}$$

In other words, the average precision score is the weighted sum of the purity reached with every prediction probability threshold. The weights are the difference between the recall of that threshold and the previous threshold. The average precision score is equivalent to the AUCPR for infinitesimally small recall steps. The purity-efficiency curve is interesting because it is desirable to maintain a high purity over the whole signal efficiency range. The ROC curve plots the signal efficiency against the background efficiency (false positive rate). For both curves, an AUC value close to 1 means that the classifier is good. The AUC makes comparing models easier. The number of trees in a model determines the complexity of a model. One can set it manually before training or let the early stopping function do the work. If the latter receives an integer n, it stops the training if the model does not improve for n rounds. The improvement depends on the evaluation metric. All computations were done on a laptop with an 4th generation Intel i7 processor and 24GB RAM. The hyperparameter search consumed 54.71 hours to test all 4860 hyperparameter combinations for the proton-proton model. Another 51.38 hours passed for the check of 4860 hyperparameters of the lead-lead model. One can choose between L1 and L2 regularization. Comparisons at the early stage of this project indicated that the L1 regularization fits the proton-proton model better. Analogously, the L2 regularization matches the lead-lead model better. The two tables below show the tested hyperparameters and the resulting optimal parameters.

Parameter	**tested Parameters**	**optimal parameter**
max_depth	2, 3, 4, 5	4
min_child_weight	1, 3, 5	5
gamma	0.1, 1, 10	10
subsample	0.8, 0.9, 1	1
colsample_bytree	0.5, 0.6, 0.7	0.5
reg_alpha	0.1, 1, 10	0.1

Table 3.3: Hyperparameter search for the model of the proton-proton collision system.

Different optimal parameters are found for the model of the lead-lead collision system:

Parameter	**tested Parameters**	**optimal parameter**
max_depth	2, 3, 4, 5	3
min_child_weight	1, 3, 5	3
gamma	0.1, 1, 10	10
subsample	0.8, 0.9, 1	0.9
colsample_bytree	0.5, 0.6, 0.7	0.6
reg_lambda	0.1, 1, 10	1

Table 3.4: Hyperparameter search for the model of the lead-lead collision system.

After the hyperparameters search, proton-proton and lead-lead models were trained using the XGBoost algorithm. As a reference, one lead-lead model trained with default hyperparameters. The appendix A shows how the early stopped proton-proton model was trained. The proton-proton training set contains 11.465.463 V^0-candidates. With this dataset, the simple model trained 46 minutes until it reached 200 members. For further reference, this model receives the name *pp2021cc*. The complex model trained until the AUC value did not increase for 50

rounds and is called *pp2021*. It took about 113 minutes to train the model to a size of 519 ensemble members. Analogously, the other two models were trained in the lead-lead collision system with 910.158 data instances. Because the dataset is a lot smaller, the training is also faster. The first model *pb2021cc* was trained 2 minutes and 21 seconds until it reached 200 members. Similarly, the training of the second model *pb2021* took 3 minutes and 33 seconds until the AUC did not increase for 50 iterations. The early stopping stopped the training when the model reached 300 members. Analogously, the reference lead-lead model was trained until the AUC value did not increase for 50 rounds. In contrast to the other models, this one used default hyperparameters. With its 292 members, it almost has the same size compared to the early stopped optimized model *pb2021*. The training took 3 minutes and 46 seconds.

3.3 Feature importance

After the training, the user can examine the feature importance for each model. There are different metrics to measure the feature importance. The XGBoost library provides the metrics *gain*, *weight*, *coverage*, *total_gain* and *total_cover*. The subsequent plots will only use the first two metrics.

Weight importance

The XGBoost ensemble consists of many decision trees. Each tree is a series of decisions. It starts with a decision based on a feature value. At this point, it splits into two other nodes. Each node is a decision, which depends on a feature. The weight describes the abundance of a feature in the ensemble. If a feature has high weight importance, the model often uses it to predict the identity of V^0-candidates. Put in another way, the weight of a feature is the proportion of the feature occurrence in the model.

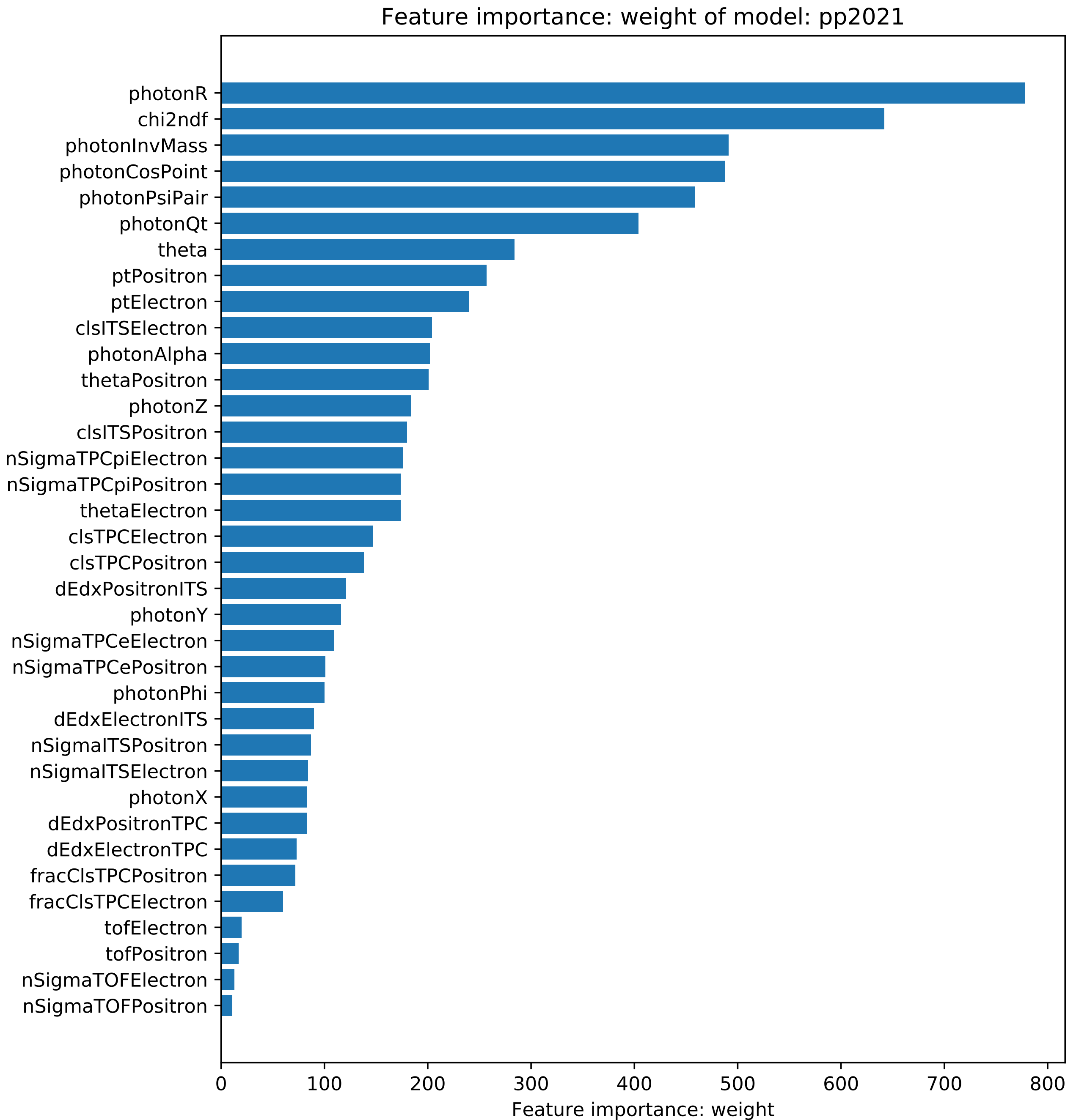


Figure 3.4: Weight feature importance of the proton-proton model *pp2021*.

Figure 3.4 shows the weight feature importance of the proton-proton model. As it seems, most of the time, the model uses the photon parameter *photonR*, which is the distance from the primary vertex at which the pair production occurs. Another important feature concerning the weight is the χ^2-test of the V^0 fit. That makes sense because combinatorial background candidates only happen to have a V-shaped trajectory. It is unlikely that they reproduce the exact V-shaped path of the V^0 daughter particles because they do not originate from the same mother particle. Other features regarding the mother particle are also important. Those features are, for instance, the invariant mass $photonInvMass$ and the cosine of the pointing angle $photonCosPoint$. Because the photon has an invariant mass of zero, it should be easy to distinguish it from particles with mass. The figure also indicates that the TOF-related features of electrons and positrons occur less often in the ensemble.

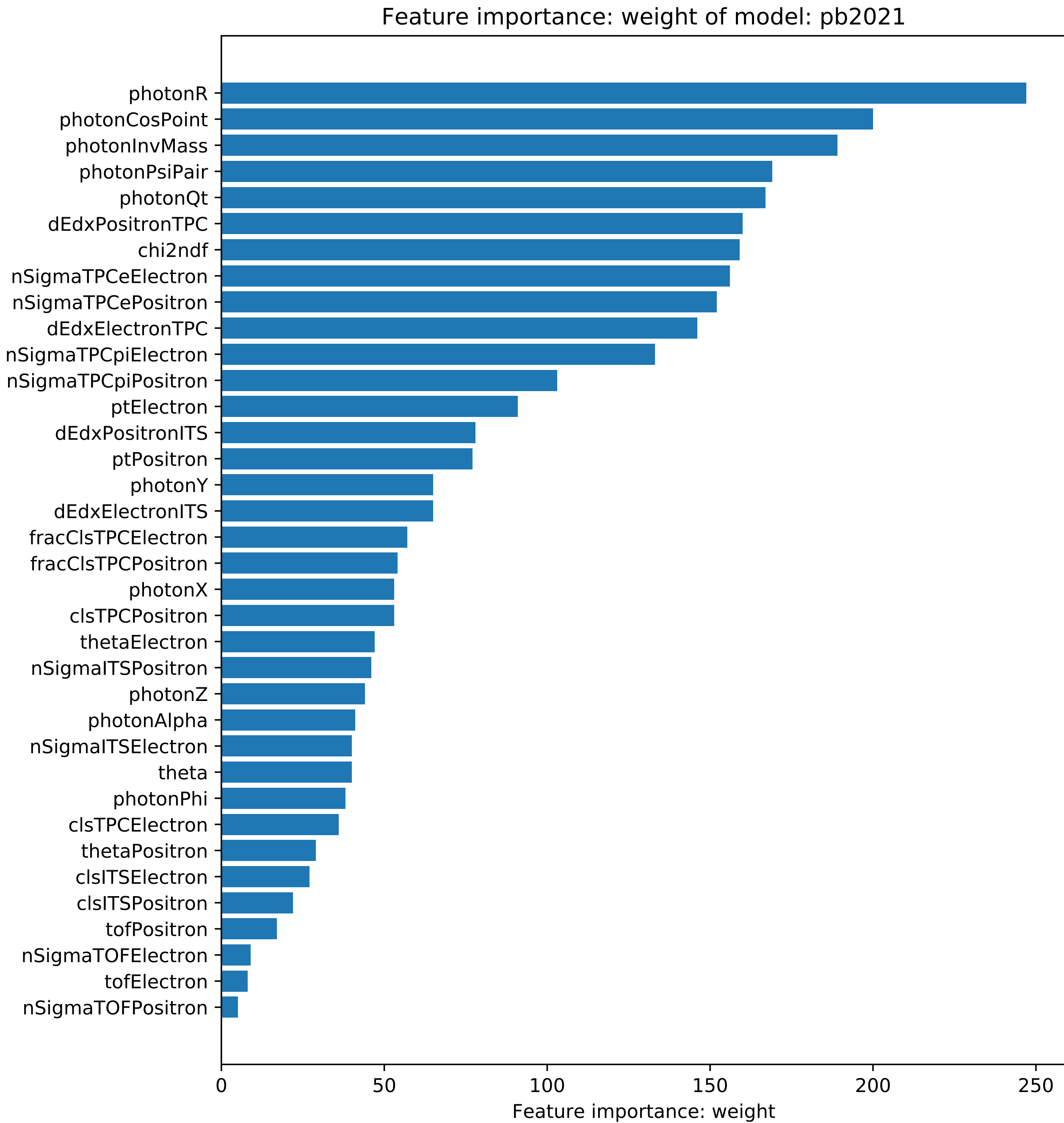


Figure 3.5: Weight feature importance of the lead-lead model *pb2021*.

In contrast to the proton-proton model, the lead-lead model does not use the χ^2-test feature as often. But similar to the proton-proton model, the lead-lead model still mostly uses *photonR*, the distance of the photon conversion from the primary vertex. Also, the photon features have high weight importance like before. Again, the time-of-flight features have low weight importance.

Gain importance

The gain importance is the contribution of a feature to the performance of a model. For instance, if the accuracy of a model increases significantly after the use of a specific feature, it has high gain importance. However, a feature can have high gain importance even if it has low weight importance. That is the case for a rare but informative detector response.

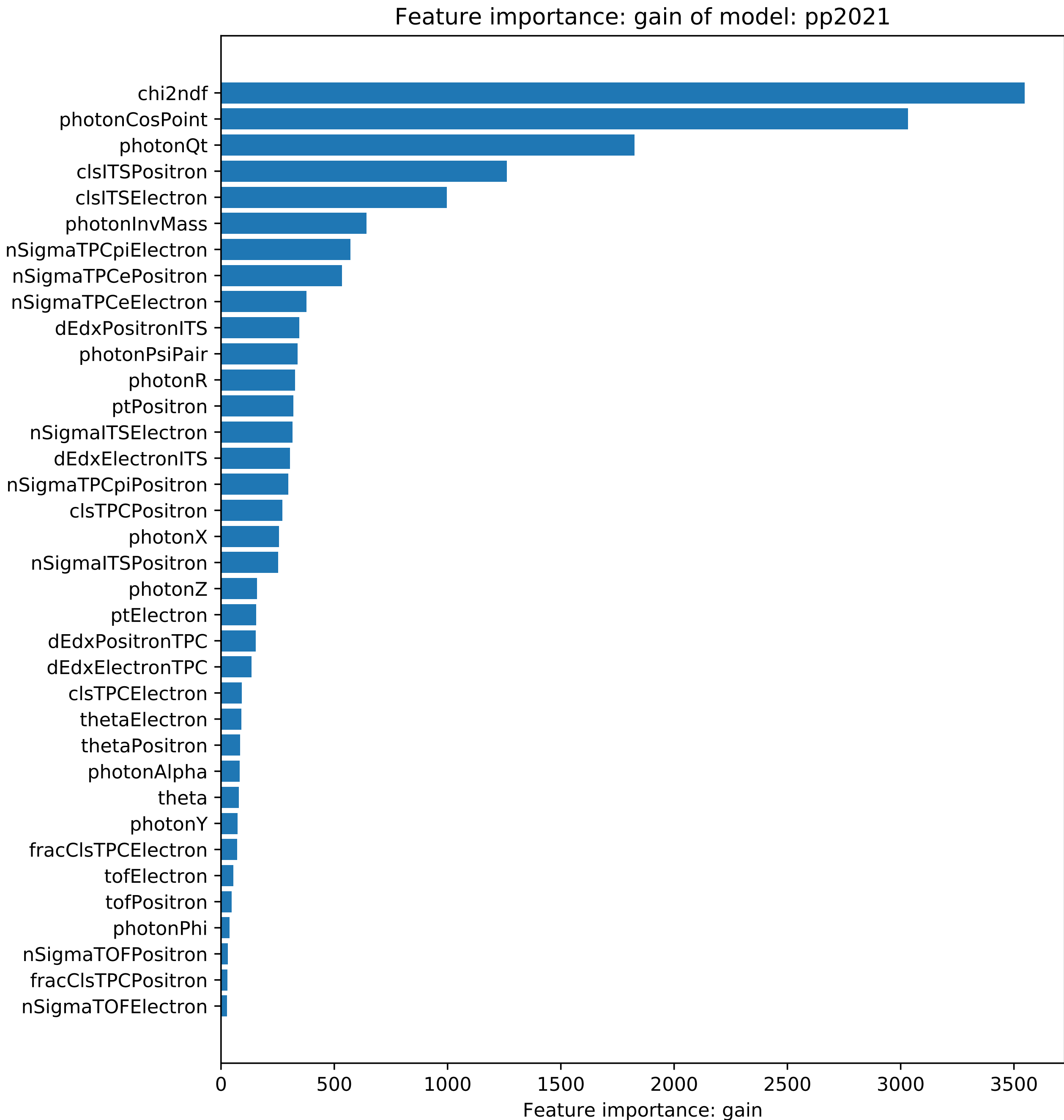


Figure 3.6: Gain feature importance of the proton-proton model *pp2021*.

Figure 3.6 depicts the gain feature importance of the proton-proton model. The importance ranking is different from the ranking in the weight importance plot. Here, the χ^2-test feature has a higher importance to the proton-proton model. That is reasonable because the χ^2 value describes how good a candidate fits as a V^0 particle. Combinatorial candidates, which do not have the same mother particle and thus, are randomly identified as a V^0 candidate, will have a worse χ^2 value. Moreover, some features related to the mother particle became less important. Only the features *photonCosPoint* and *photonQt* stayed important. Note that *photonCosPoint* is the cosine of the pointing angle of the mother particle. Therefore, it would make sense that it is an important feature. One would expect that the conversion photon has a pointing angle close to zero and thus a cosine close to one. Combinatorial candidates do not originate from the same mother particle. It is unlikely that their pointing angle is zero. Remember that *photonQt* is the momentum transfer in the transverse direction. The transverse momentum transfer of a conversion photon should be nearly zero. The randomly combined particles will not have a

meaningful value for the transverse momentum transfer. Unlike the previous weight importance, the gain importance gives the number of clusters triggered by positrons and electrons in the ITS *clsITSPositron* and *clsITSElectron* higher importance. These two features do not occur that often but contribute more to the performance of the proton-proton model. The distance to the primary vertex *photonR* has high weight importance but low gain importance for the proton-proton model. A way to interpret this is that it has a low relevance regarding the prediction, which means that one could leave out this feature without a substantial loss in model performance. Similar to the weight importance plot, the gain importance plot also shows the low importance of the time-of-flight features.

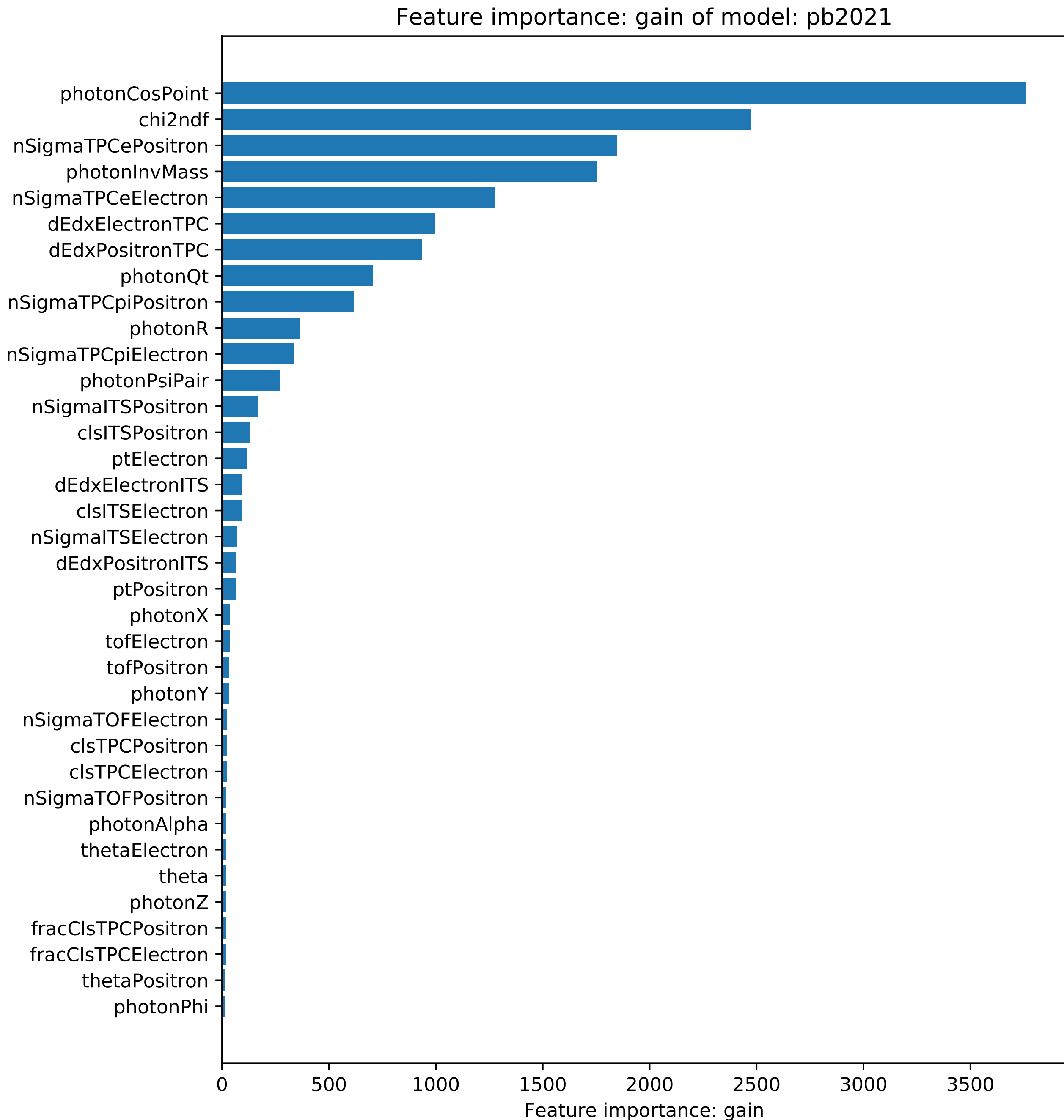


Figure 3.7: Gain feature importance of the lead-lead model.

Surprisingly, the χ^2-test feature contributes far less to the performance of the lead-lead model than the cosine of the pointing angle *photonCosPoint*. The deviation to the electron band in the energy loss plot of the TPC *nSigmaTPCePosition* and the invariant mass of the mother

particle *photonInvMass* also contribute a lot to the performance of the lead-lead model. The usage of the positron deviation in the TPC energy loss plot is reasonable because it can help to distinguish electrons and positrons from pions. Figure 3.8 shows an example for the energy loss plot of the TPC. It makes sense that an data instance, which is further away from the electron band, could more likely be another particle, such as a pion.

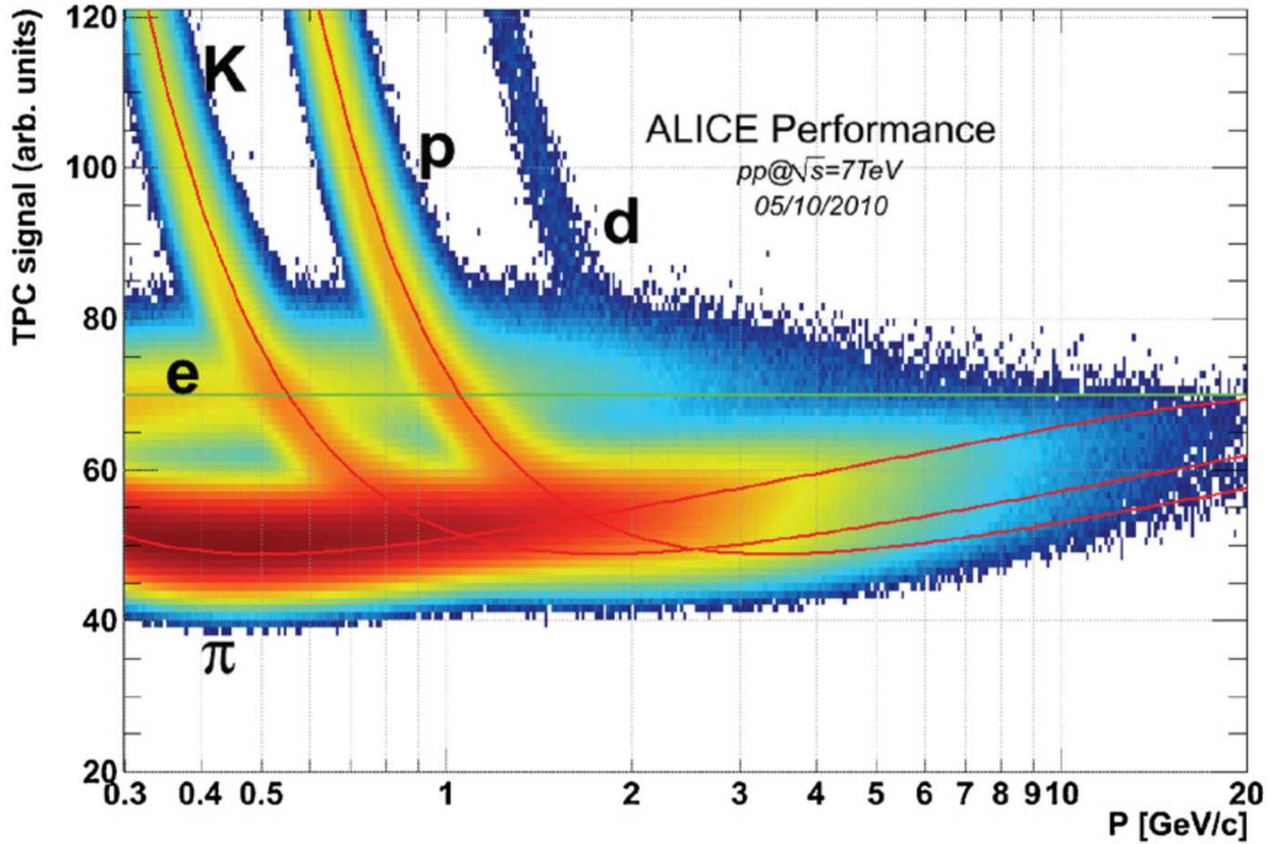


Figure 3.8: TPC: Energy loss vs. momentum [18].

The previous section already introduced the feature importances and the different rankings for each model. A few features share a large proportion of importance. Thus, the next part will examine the distribution of primary photons and background regarding each feature. Each plot uses the same 500000 data instances of the corresponding collision system.
Figure 3.9 shows the three most important features for the proton-proton model for each metric. More precisely, it shows histograms, where one can see the photon and background distribution regarding a feature. The histogram is normalized, such that the sum of all bars is equal to 100%. Analogously, figure 3.9 contains the histograms for the lead-lead model. Other feature histograms can be found in the attachment A.

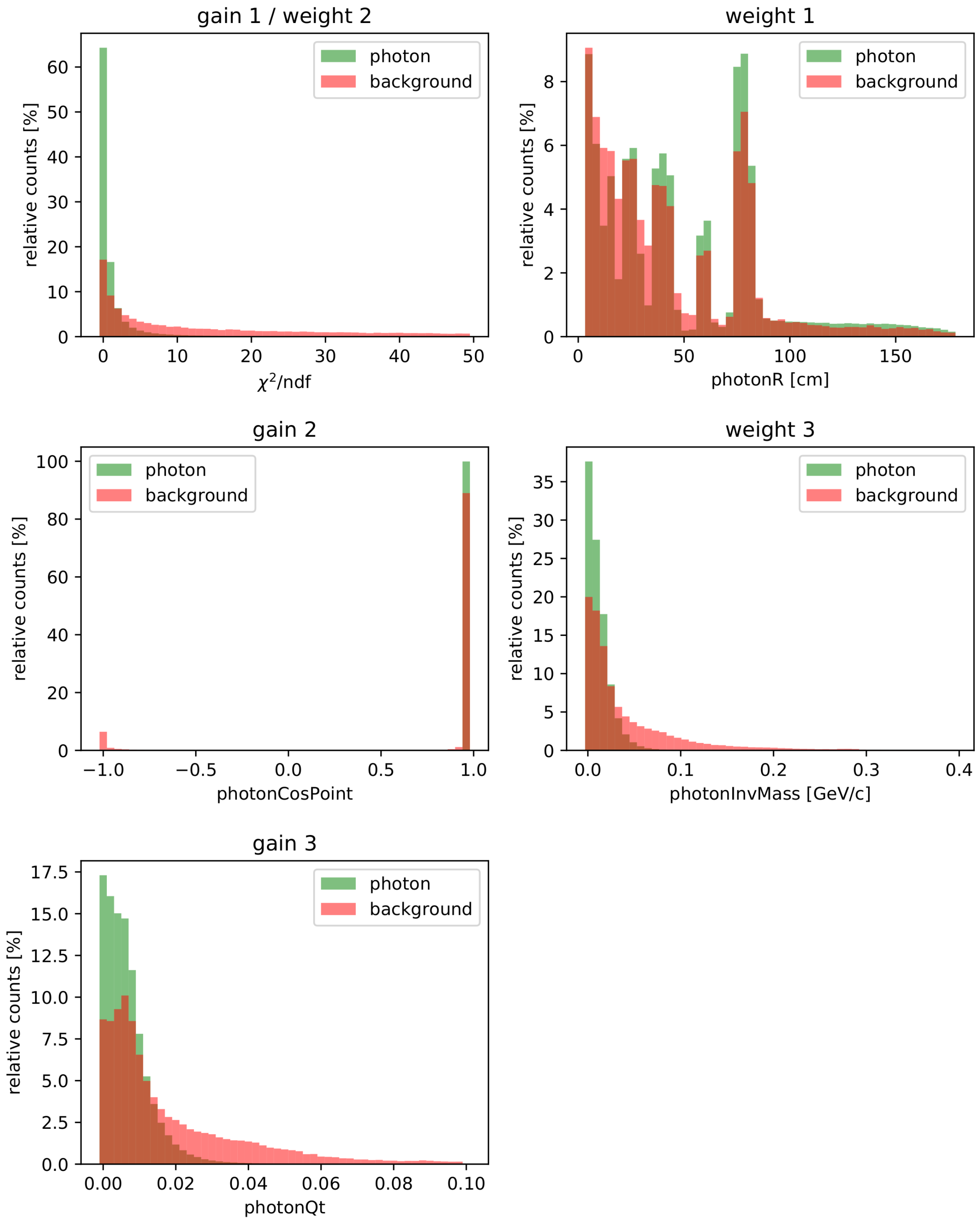


Figure 3.9: Important features for the proton-proton model *pp2021*.

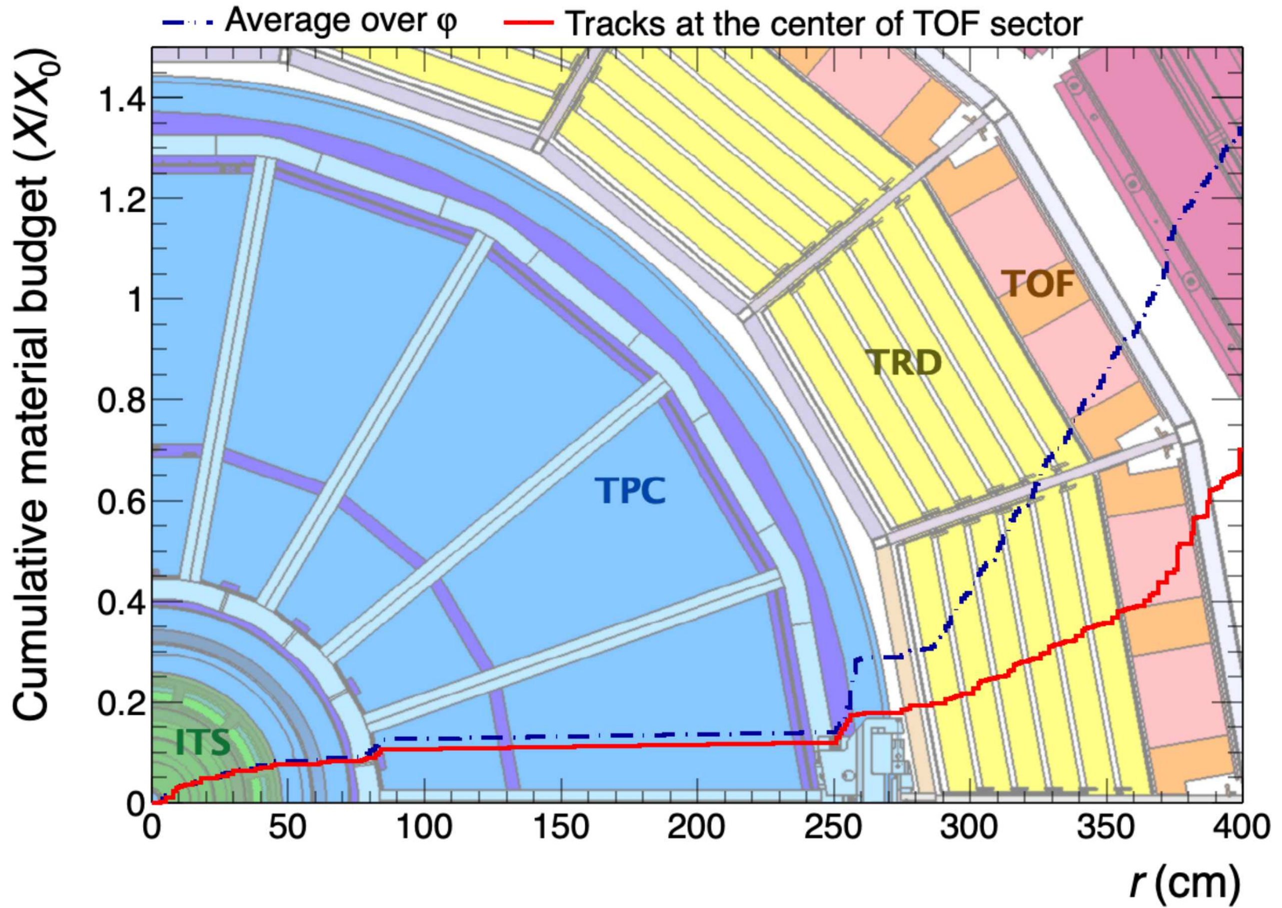


Figure 3.10: Material budget of ALICE as a function of the distance from the beam line [19].

As mentioned before, the radial distance from the primary vertex *photonR* has high weight importance for the proton-proton model. The histogram shows a distribution one would expect. One can observe peaks at certain distances. That is because the material budget at these positions is high. In contrast to the photons, the background is also measured at other distances. The distribution of the background is less dependent on the material budget. The model could learn that the background distribution does not drop between the photon distribution peaks. Figure 3.10 shows the cumulative material budget as a function of the distance from the primary vertex. At around 80cm, there is a sudden increase in material budget. After that, no additional peaks can be seen in the histogram in figure 3.9. The reason is that the gaseous volume of the TPC starts there. Moreover, there is a cut at 180cm such that there are no V^0-decay or conversion signals after that point. However, the decay or conversion products still can reach the TRD and the TOF. But if one takes a look at the figures regarding the TOF responses in the appendix A, one observes that more than 80% of the data is contained in a peak at around 20000 ps. That peak might represent the uninformative signals. The figures A.19 and A.27 show the useful signals in a restricted interval. These informative TOF signals only have a low abundance. Thus, the TOF features have such low importance. Nonetheless, it is not apparent why the model frequently checks the distance from the primary vertex *photonR*. The distributions are not distinct enough to make a simple cut for classification. But this is expected because this feature has a low gain feature importance. On the other hand, the gain feature importance gives the fit goodness χ^2 high value. One can see that the photons occur at values close to zero, while the background distribution is more smeared out. In the subsequent gain plot, the background also happens to have a pointing angle of 180 degrees. That makes sense if the V^0-finder mistakes particles, which travel towards each other as a V^0-pair. That could be the case for combinatorial candidates. Therefore it is simple to identify a part of

the background with this histogram. As one would expect, the invariant mass of the mother particle helps with the prediction. It should be zero for photons. As can be seen in that histogram, the distributions slightly overlap such that the informative value is lower. Putting both together that the photon has an invariant mass of zero and the overlapping distributions, it makes sense that this feature is more important regarding the weight. One could perform a cut slightly below 0.1 GeV/c. The relative momentum transfer *photonQt* is the third most important feature regarding the gain, which is not surprising. Because the momentum must be conserved and the photon has no mass, the momentum transfer is zero. Other combinatorial candidates are random combinations and lead to random meaningless momentum transfers. Decays of heavier particles, such as the decay of the neutral kaon K_s^0 into pions, have a greater momentum transfer.

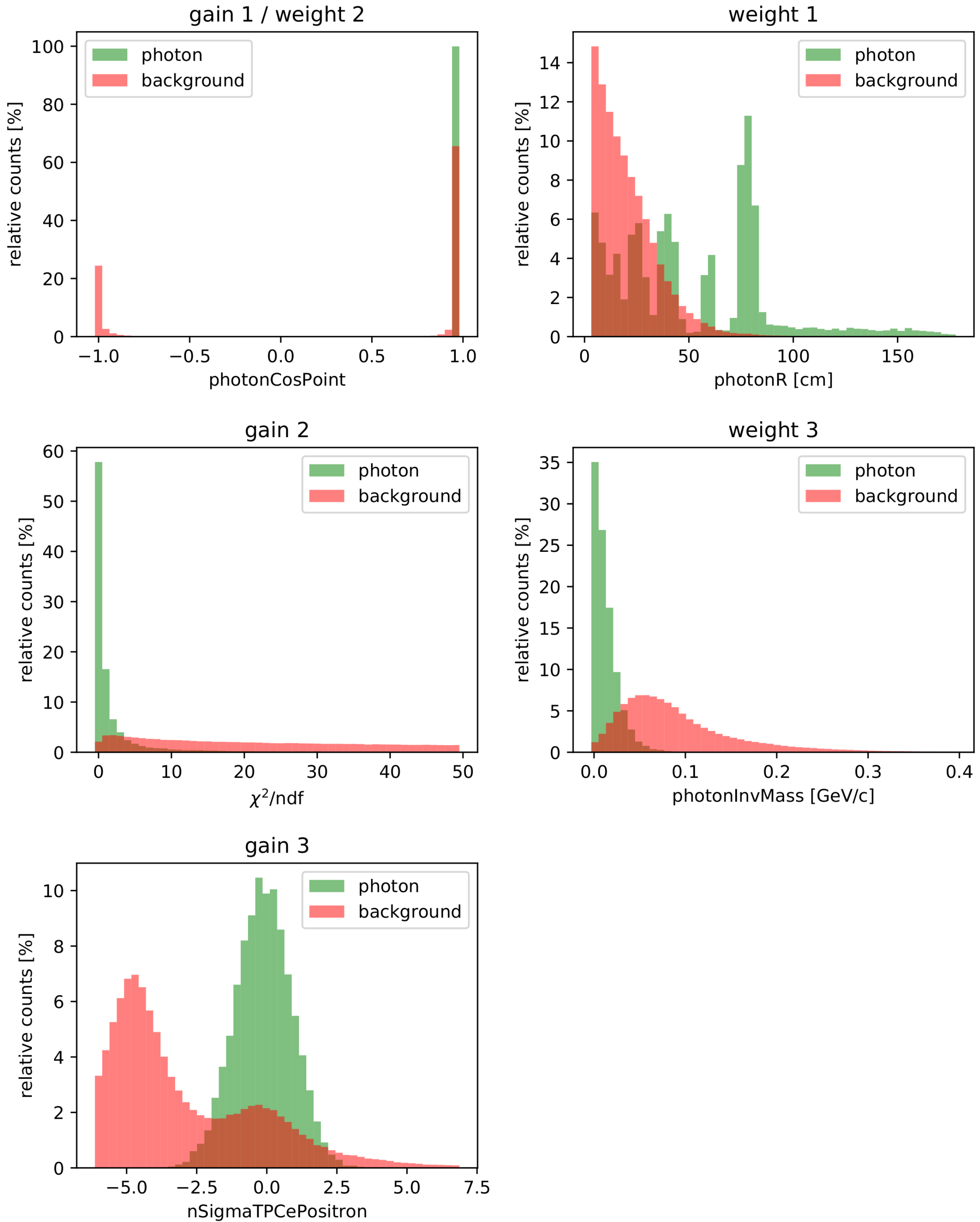


Figure 3.11: Important features for the lead-lead model *pb2021*.

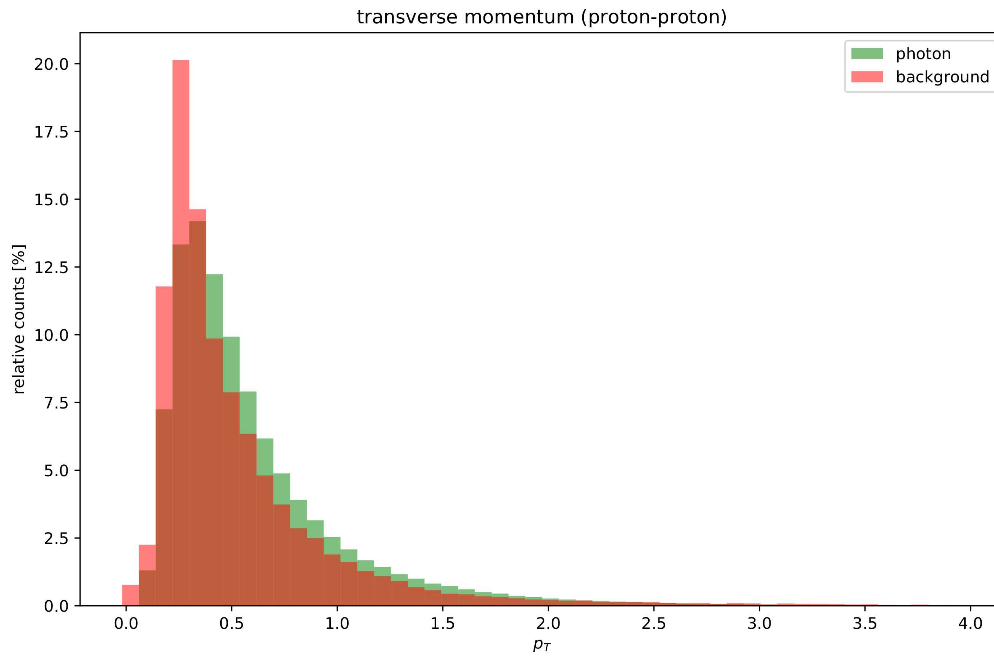
In contrast to the proton-proton collision system, the lead-lead collision system shows a better separable *photonR* (distance from the primary vertex) histogram. Here, the background steadily decreases with the distance from the primary vertex. It has no peaks at the high material-budget positions. Meanwhile, the primary photons still convert at these high material-budget positions. It seems that a cut at around 50cm is possible. Although it appears informative, there were other features with higher gain importance. Thus, it was only most important regarding the

weight. It is reasonable that this feature occurs regularly in the ensemble. The χ^2 histogram shows that the background is smeared out even more than before. One could easily split photons from the background by cutting at around a value of 4. It makes sense that it is one of the most important features concerning the gain. When compared to the proton-proton collision system, the amount of background at $\cos(\Theta_p) \approx -1$ increased. It still makes it simple to discard more than 20% of the combinatorial candidates. That might be the reason why it is important regarding both metrics. Converted photons result in an electron and a positron. In the energy loss plot of the TPC, it is possible to distinguish electrons and positrons from other particles like pions. The figure shows that most of the positrons of the primary photons are within the 3σ range. By performing a cut at about 2.5σ, a lot of background can be separated. It is reasonable that it has high gain importance to the lead-lead model. In the histogram of the invariant mass, the distribution of photons and background can easily be separated. As one would expect, the photons have a value close to zero. By cutting at around 0.025 GeV/c, one can discard a large amount of background.

3.4 Evaluation of XGBoost models

As mentioned before, the models are trained without the transverse momentum p_T feature. The idea is that the models should perform well across the whole transverse momentum range. In the subsequent sections, the transverse momentum range will be divided into eight intervals, such that the model performance is visible across the momentum interval. The size of these subsets decreases with higher momentum, which is shown in figure 3.12a for the proton-proton dataset and figure 3.12b for the lead-lead dataset.

(a) proton-proton dataset.



(b) lead-lead dataset.

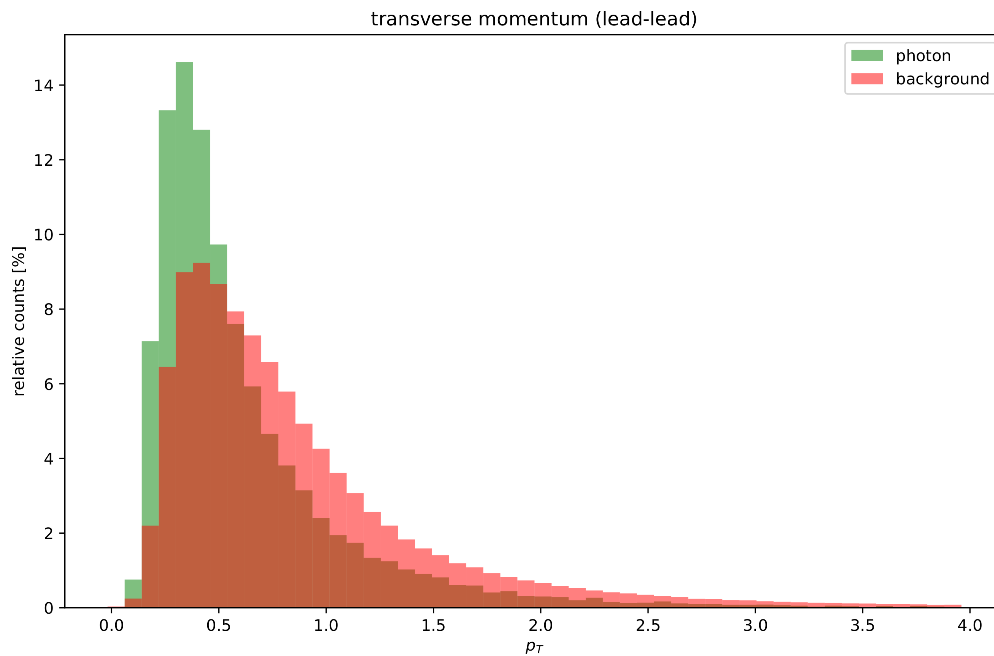


Figure 3.12: Transverse momentum histograms.

3.4.1 Comparison with cut-based model

This section pursues the comparison of the XGBoost and the cut-based model. The focus lies on the signal efficiency of the models as a function of transverse momentum. For this purpose, the classification threshold is adjusted such that the purity of the XGBoost models is equal to that of the cut-based model. Based on that, an evaluation method computes the signal efficiency within different transverse momentum p_T intervals. Both the method to adjust the purity and the evaluation method are in the attachment A. The p_T range starts at 0 GeV/c and ends at 4 GeV/c and is split into 8 intervals.

Proton-proton collision system

The proton-proton collision system serves as a reference for the lead-lead collision system. First, the thresholds are adjusted such that the purity of the XGBoost and the cut-based model are equal.

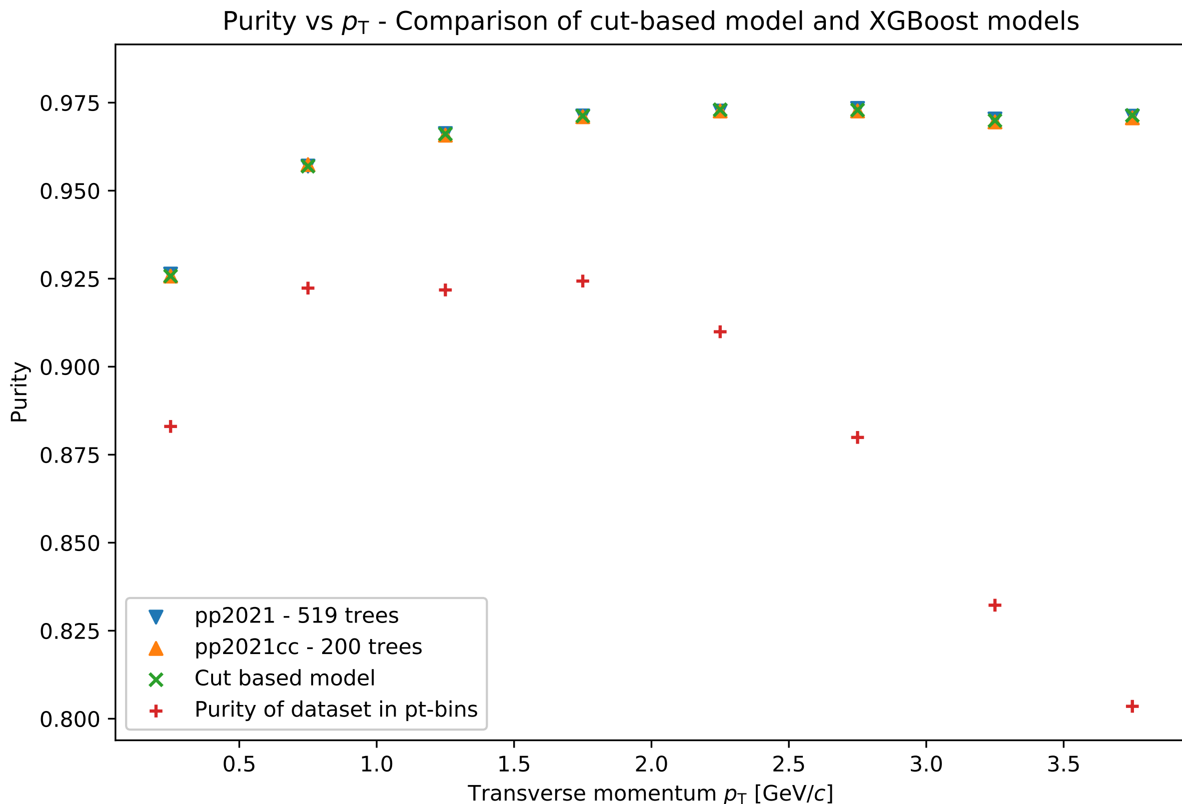


Figure 3.13: Adjusting the purity of the XGBoost models to the purity of the cut-based model in each transverse momentum bin. (proton-proton collision system)

Figure 3.13 shows that the purity of both XGBoost models is equal to that of the cut-based model in each transverse momentum bin. The red data points show the purity of the data subset in each bin. According to equation 2.4, that is the number of correctly classified primary photons divided by the number of predicted primary photons, which is the sum of correctly and falsely classified photons. One can observe that it is maximal between 0.5 and 2 GeV/c. It decreases with higher transverse momentum, but overall, the purity of the dataset is already high and constantly above 80%. Therefore, it is easier for the classifier to perform well, as they could classify every input as a photon and still reach a high purity. The cut-based model has

a lower purity at lower transverse momentum (≈ 92.5%) but slowly rises to a purity of about 97%.

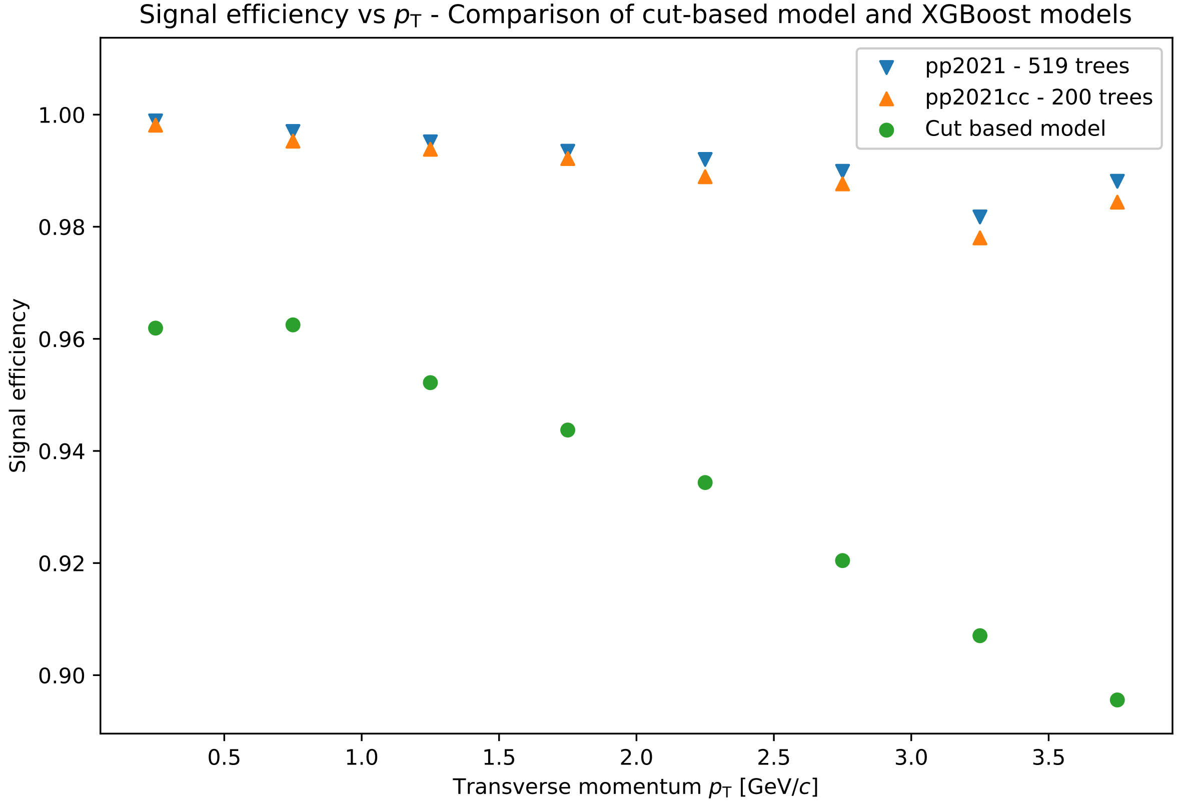


Figure 3.14: The signal efficiency of the models evaluated in each transverse momentum bin after the equalized purity. (proton-proton collision system)

From the equation 2.5, it is known that the signal efficiency is the number of correctly classified primary photons divided by the number of real primary photons. As can be seen in figure 3.14, all models achieve signal efficiencies above 90%. With higher transverse momentum p_T, the signal efficiency decreases. But while the cut-based model loses around 6%, the XGBoost models only lose a maximum of 2% signal efficiency. Compared to the cut-based model, the signal efficiency of the XGBoost models is higher in all p_T intervals. At low transverse momentum, the difference is about 4%, and at higher momentum, the difference rises to 9%. Both XGBoost models performed similarly well, but the smaller ensemble *pp2021cc* always had a slightly lower signal efficiency. With higher transverse momentum, the difference even increased. In other words, the XGBoost models enable us to extract more primary photons from the dataset than the cut-based model while still offering the same purity.

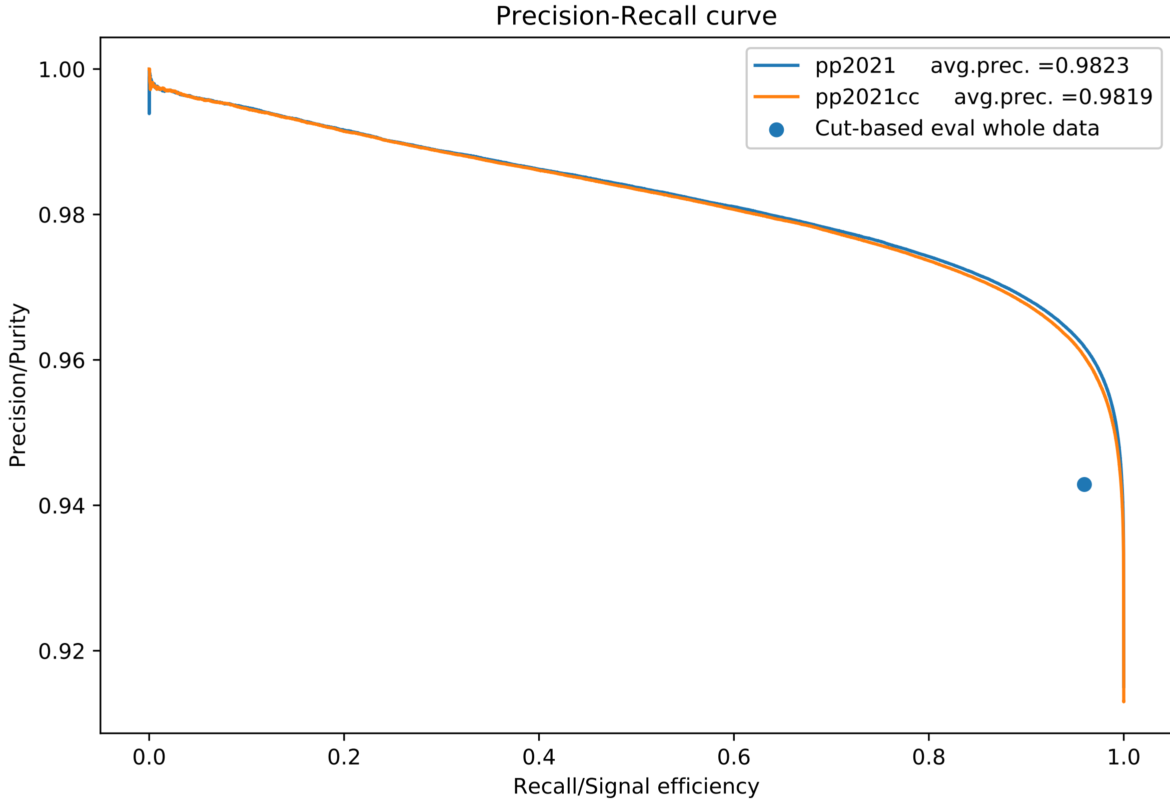


Figure 3.15: Purity-efficiency curves. (proton-proton collision system)

Figure 3.15 shows the purity-efficiency curve. Looking at the purity-efficiency curve in this figure, one can see the achieved purity of the classifiers at different signal efficiency values. The evaluation uses the whole validation dataset. It is easier to compare the models by their average precision score than by their efficiency at different purity values. Both XGBoost models perform well and have an average precision score of around 0.98. The blue dot in the figure represents the efficiency and purity that the cut-based model reached with the evaluation dataset. It confirms the previous observations that the XGBoost models can reach a higher signal efficiency than the cut-based model at the same purity. The difference between the XGBoost model with only 200 trees *pp2021cc* and the early stopped model with 519 trees *pp2021* is marginal. If rounded to the third decimal place, the average precision score would be the same. The difference is therefore negligible, and the models perform similarly well.

Lead-lead collision system

In figure 3.16, one can see that the purity of the lead-lead collision system dataset is lower compared to the purity of the proton-proton dataset. It is below 10% and even decreases with higher transverse momentum. The model could classify all candidates as background and thereby have a correct prediction above 90% of the time. But this would not fulfill the goal to identify photons among V^0-candidates. As one would also expect, the cut-based model cannot reach the same high purity as in the proton-proton dataset anymore. Between a transverse momentum of 0.75 GeV/c and 3.25 GeV/c, the cut-based model still reaches a purity between $70 - 80\%$. Because the subsets in each bin are also smaller than in the proton-proton dataset, it was harder to find a threshold such that the XGBoost models reach the same purity. A cut is performed on the prediction probability. Thus, if many candidates have the same probability of being a primary photon, the probability threshold cannot divide the instances. That makes

it impossible to achieve the same purity as the cut-based model.

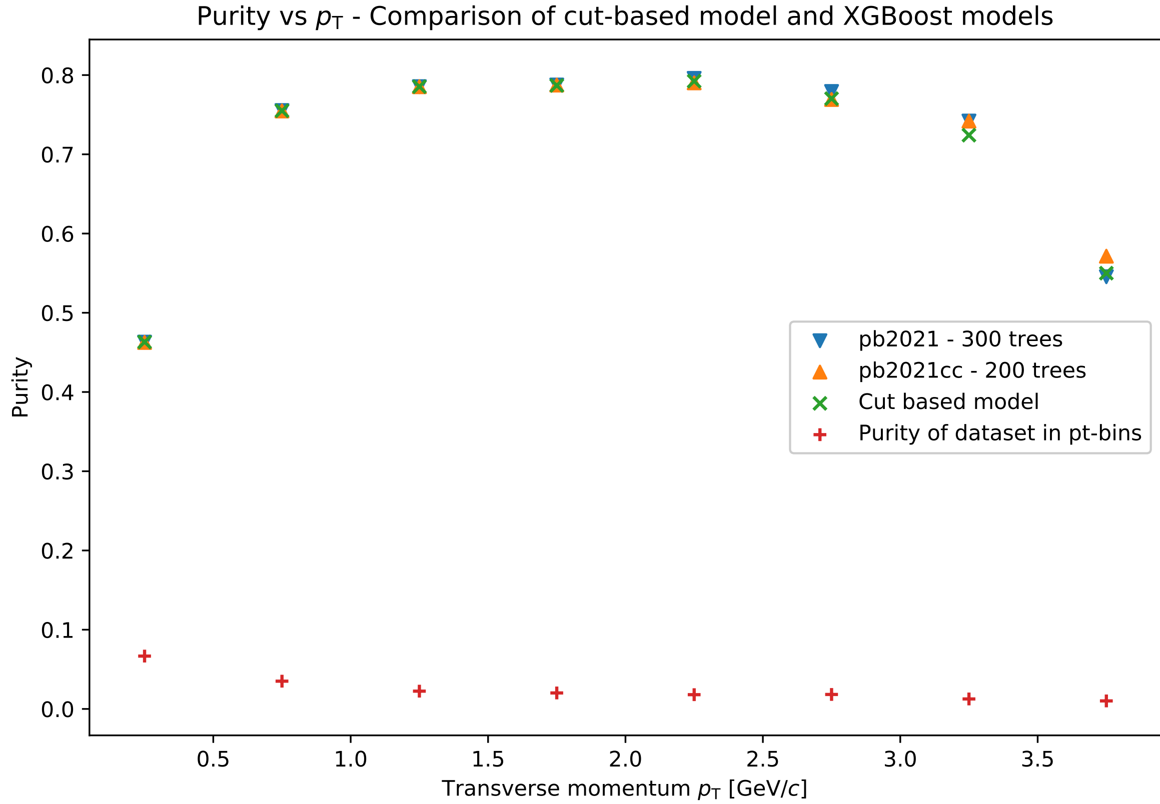


Figure 3.16: Adjusting the purity of the XGBoost models to the purity of the cut-based model in each transverse momentum bin. (lead-lead collision system)

Figure 3.17 shows that the XGBoost models perform very well and reach a signal efficiency very close to 1 most of the time, whereas the cut-based model only achieves a signal efficiency between $91-95\%$. The XGBoost model's performance does not deviate much over the range of transverse momentum and instead holds up a constant efficiency of $99-100\%$. At a transverse momentum between 0.5 and 1.5 GeV/c the XGBoost models perform slightly worse.

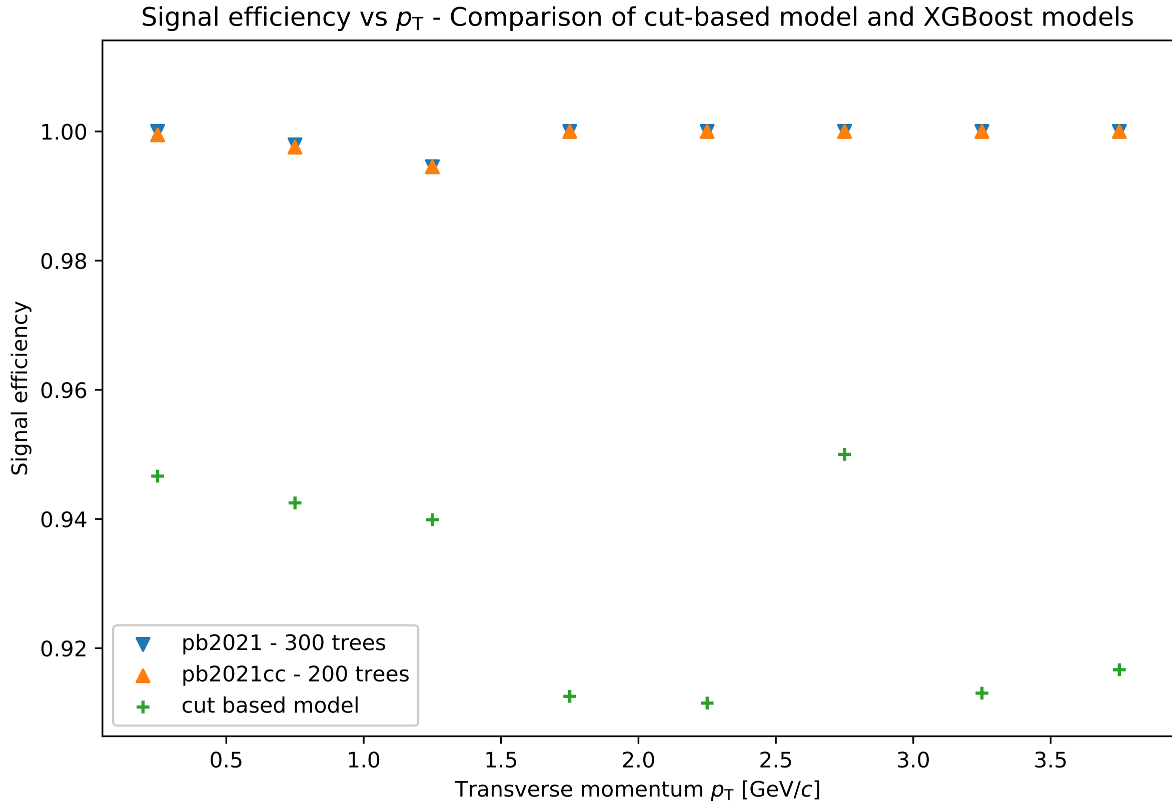


Figure 3.17: The signal efficiency of the models evaluated in each transverse momentum bin after the equalized purity. (lead-lead collision system)

Similar to the purity-efficiency curve of the proton-proton dataset models, the PR curve in figure 3.18 shows that both XGBoost models perform better than the cut-based model. With an average precision of around 96%, both models have a high average precision score, although they are lower than the score of the proton-proton dataset models. Again, the difference is only marginal. But this time, the early stopped model *pb2021* has a higher score than the 200 trees model *pb2021cc*, even when rounded to the third decimal place.

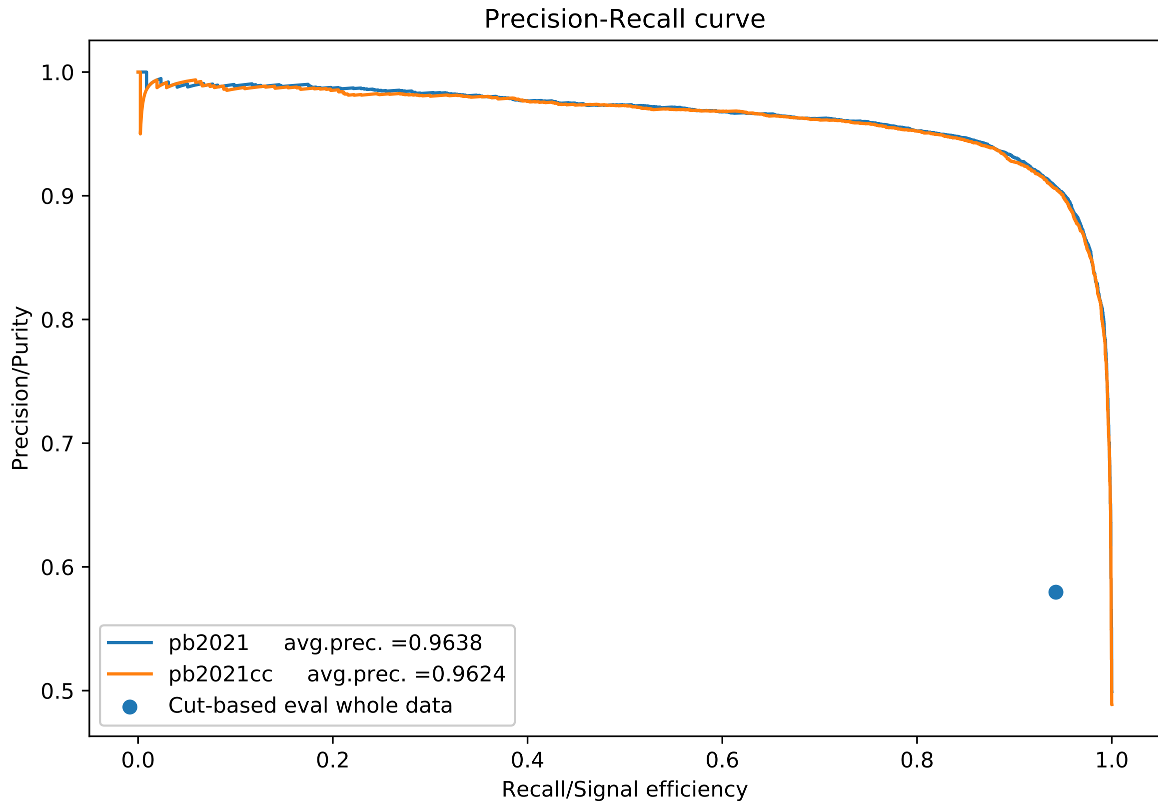


Figure 3.18: Purity-efficiency curves. (lead-lead collision system)

These figures showed that the XGBoost models performed better than the cut-based model. Especially with increasing transverse momentum, the efficiency of the cut-based model dropped in both collision systems, while the XGBoost models held up a high signal efficiency. Moreover, the difference between the two XGBoost models was only marginal compared to the difference in the cut-based model.

3.4.2 Performance of XGBoost models at high purities

In practice, one would like to have very pure photon datasets. Therefore, this section documents the performance of the XGBoost models at high purity.

Proton-proton collision system

First, the thresholds are adjusted such that the classifiers achieve purities of 94%, 96%, and 98% in the proton-proton dataset.

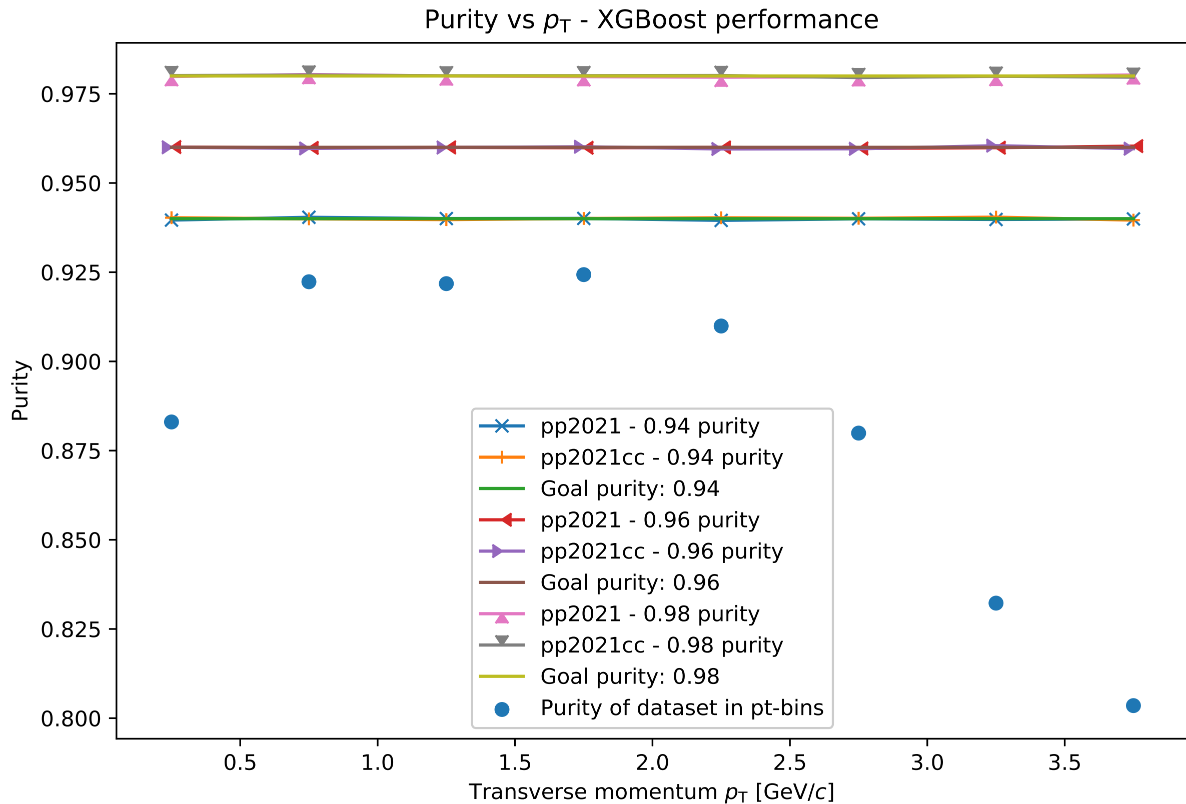


Figure 3.19: Adjusting the purity of the XGBoost models to the goal purities in each transverse momentum bin. (proton-proton collision system)

Figure 3.19 shows how well the adjustment was. There are no significant deviations from the goal purity.

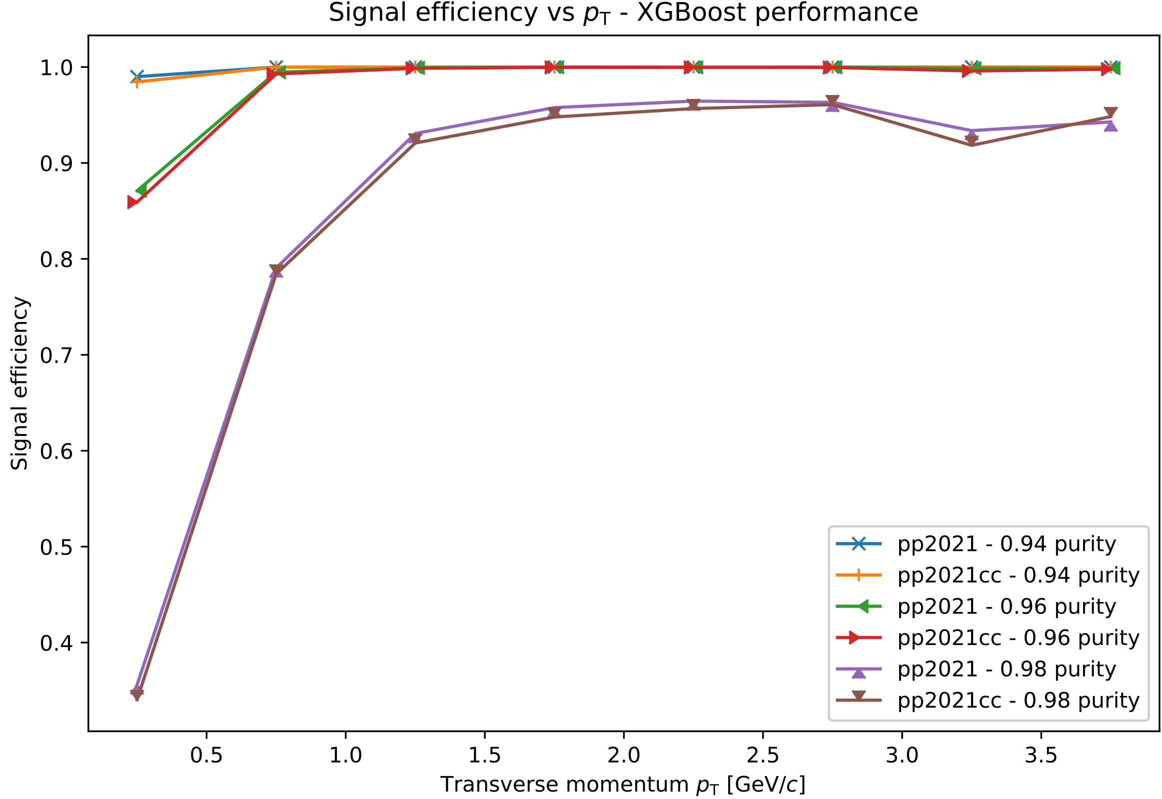


Figure 3.20: Signal efficiency of the XGBoost models at 94, 96 and 98% purity. (proton-proton collision system)

As shown in figure 3.20, the signal efficiency starts low and rises with higher transverse momentum. With a purity of 94% and 96%, the signal efficiencies are already very close to 100% after the first p_T-bin. At a purity of 98%, the models reach a signal efficiency above 90% after the second transverse momentum bin or above 1.5 GeV$/c$. It is a trade-off between purity and efficiency. But with a high purity like 96%, it is already possible to constantly reach high efficiency in the proton-proton collision system. The signal efficiency is always above 85% when the purity is set to 96%. The early stopped model *pp2021* also tends to reach higher signal efficiencies than the 200 member ensemble *pp2021cc*. While the signal efficiency difference between the two models is rather small at a purity of 94 and 96%, the signal efficiency of the *pp2021* model is visibly higher than that of the *pp2021cc* model at 98% purity. So, it seems to be a good choice to use a well-trained and optimized model for photon classification for very high purity.

Lead-lead collision system

Because the primary photons are less abundant in the lead-lead collision system, it is harder to maintain a constant high purity.

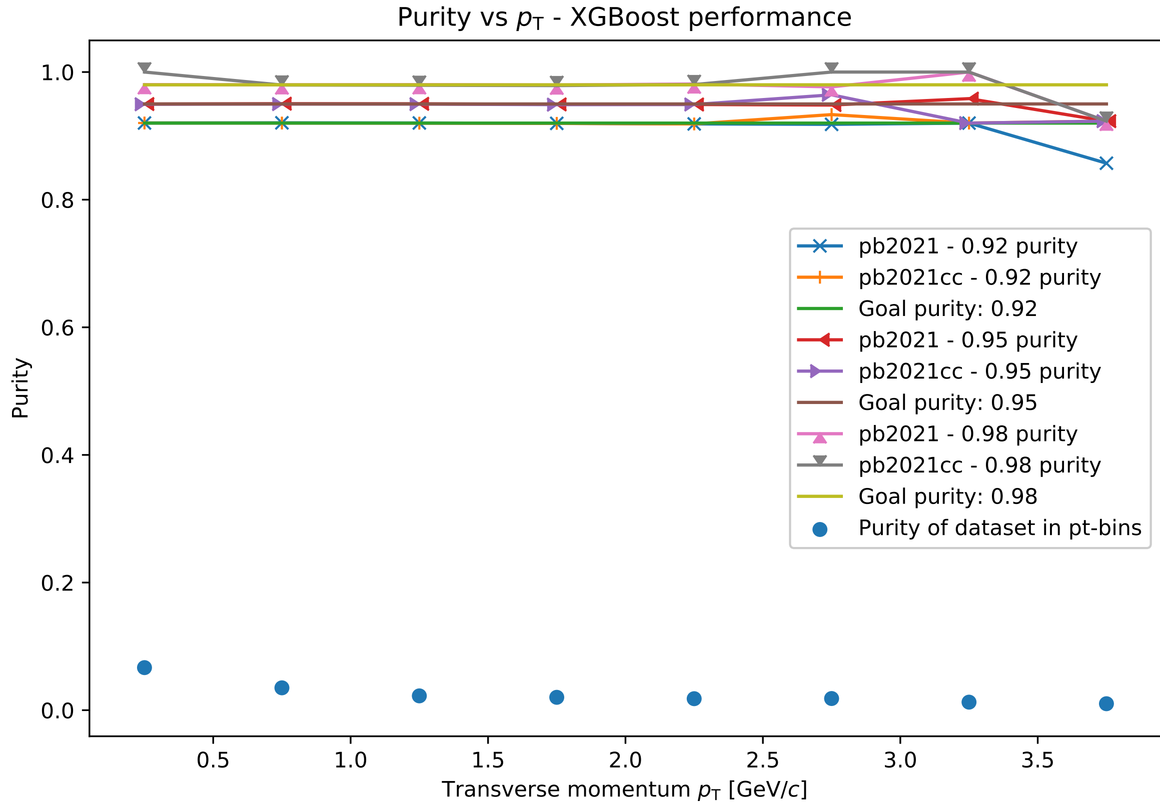


Figure 3.21: Adjusting the purity of the XGBoost models to the goal purities in each transverse momentum bin. (lead-lead collision system)

The purities deviate from the goal purities, especially at high p_T, which can be seen in figure 3.21. But this could also be caused by the small dataset. The statistical deviations have a bigger impact when the data subset happens to be small, which tends to occur at high transverse momentum. Here, the goal purities cover a broader range such that at least one model can perform constantly well. The threshold are adjusted such that the XGBoost models reach a purity of 92%, 95%, and 98%.

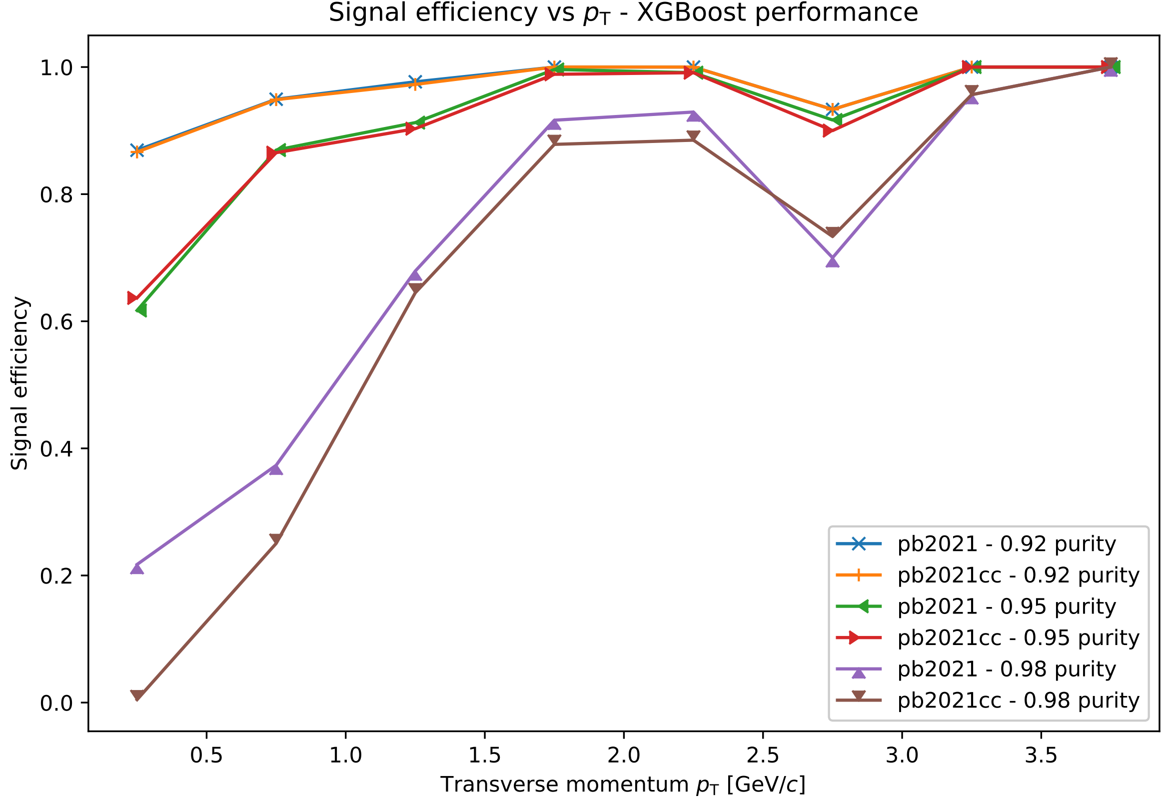


Figure 3.22: Signal efficiency of the XGBoost models at 92, 95 and 98% purity. (lead-lead collision system)

The lead-lead models do not reach signal efficiencies as high as the efficiencies of the proton-proton models. At a purity of 98%, the early stopped model *pb2021* started with an efficiency of about 20%. It seems that the model with 200 members *pb2021cc* could not find any photons such that its signal efficiency is about 0%. Independent from the purity, the signal efficiency of both models slowly rises with higher transverse momentum. Only when the transverse momentum reaches a value of around 1.25 GeV$/c$, the XGBoost models achieve a signal efficiency above 60%. The performance drops significantly at approximately 2.75 GeV$/c$ for both models at all observed purities. Until that point, the early stopped model constantly had higher efficiency. However, the 200 member model was better than the early stopped model at this point. If the purity of *pb2021cc* would be lower than that of *pb2021*, it could make sense, but it was not. A possible explanation could be that it is just a statistical deviation and not a systematic error of the models. That would require the evaluation of more data. The lead-lead models at 95% purity start at around 20% signal efficiency and already reach a signal efficiency above 80% after the first p_T-bin around 0.75 GeV$/c$. At a lower purity of 92%, the models can keep up an efficiency above 85%. The early stopped model *pb2021* tends to perform better than the 200 member model *pb2021cc*, especially at the high purity of 98%.

3.4.3 Comparison with default lead-lead model

This section compares the optimized XGBoost models to the default lead-lead model. That model was trained with default hyperparameters. Furthermore, the comparison also comprises the cut-based model. Subsequently the optimized model will be known as *pb2021* and the default model known as *pb2021_default*.
Figure A.5 in the appendix presents the signal efficiency, which the models achieved in each transverse momentum bin after the purity was adjusted to the purity of the cut-based model. The difference in this plot is marginal. The default model even reaches a higher signal efficiency than the optimized at 1.25 GeV/c. There is no significant difference with the hyperparameter search. Thus, it is sufficient to use the default hyperparameters for the training.
At the high purity of 98% the early stopped and optimized model *pb2021* performs better at lower transverse momentum below 1.5 GeV/c. At a transverse momentum between 2 and 3 GeV/c, the default model performed better than the optimized model as the efficiency of the optimized model dropped significantly in this region. The default model did not lose efficiency inside this interval. The reason for that could be that the default model is slightly less prone to overfitting and therefore generalizes better. It was not "optimized" such that its AUC score is maximal.
Figure A.7 in the appendix shows the attempt to adjust the purity of the models in each transverse momentum bin to the goal purity. The default model is slightly lower in the critical region between 2 and 3 GeV/c.

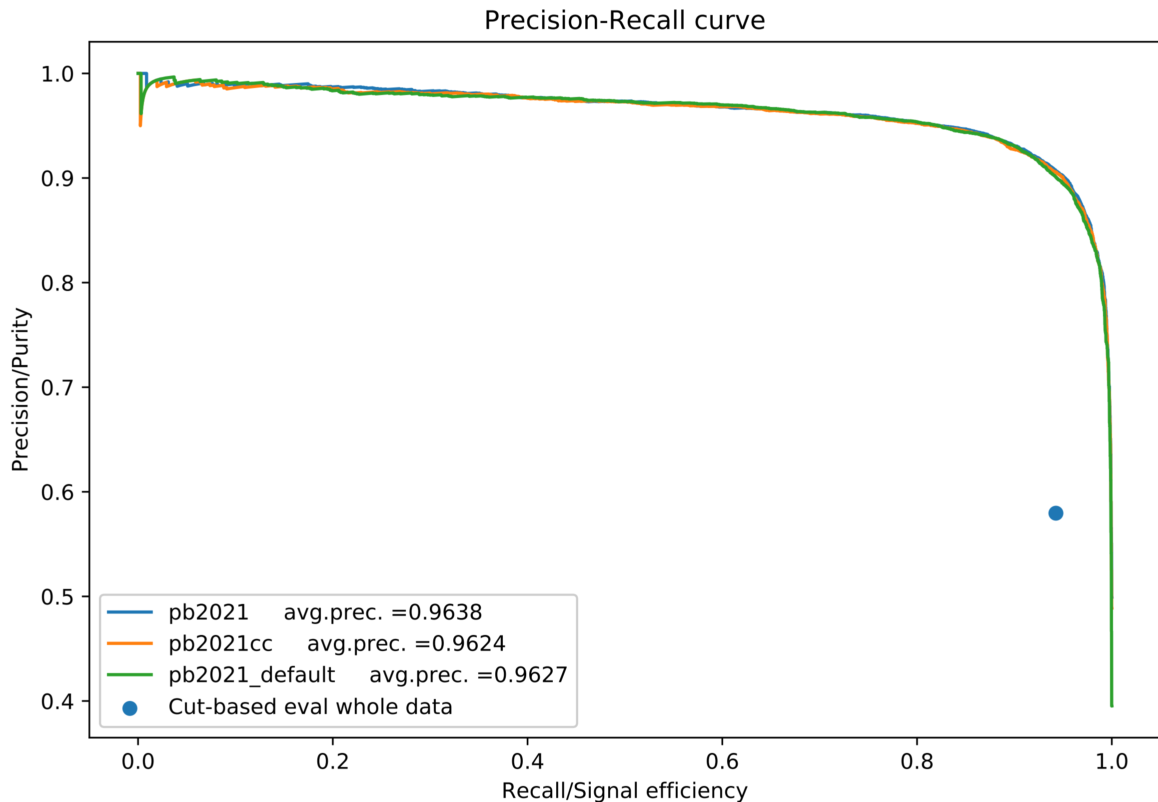


Figure 3.23: Purity-efficiency curves. (lead-lead collision system)

The default model seems to perform marginally better than the optimized models. But the PR-curve in figure 3.23 also shows this result. The optimized model has a slightly higher

average precision score than the default model. But in comparison to the cut-based model, the difference between the XGBoost models is negligible.

3.4.4 Comparison with random forest

Another less complex ensemble learning method is the random forest. In contrast to boosted decision forests, this algorithm trains trees in parallel. Trees are not created sequentially and such that the correlation is minimal. That is done by bagging and feature subsampling. Bagging is to use random subsets of the training set for each tree. Feature subsampling describes the process of using only a subsample of the features for a node split of a tree [20].

Proton-proton collision system

A random forest with 300 trees was trained without hyperparameter optimization on the proton-proton dataset. The training took 37 minutes and 48 seconds.
Figure A.8 in the appendix shows the adjustment of the purity of the random forest and XGBoost model to the purity of the cut-based model. There were no significant deviations.

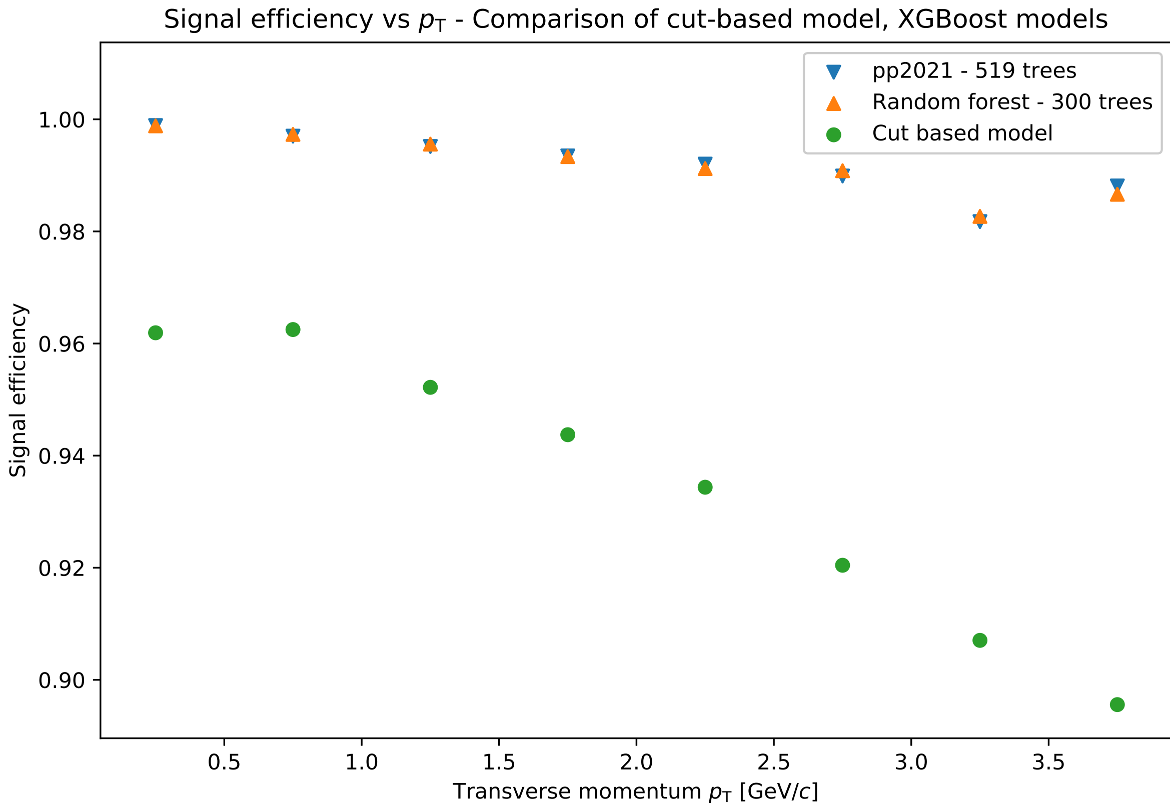


Figure 3.24: The signal efficiency of the models evaluated in each transverse momentum bin after the equalized purity. (proton-proton collision system)

The difference between the random forest and the XGBoost model *pp2021* is almost not noticeable. It also achieved a signal efficiency between 98 and 100%. The performance of the random forest and the XGBoost model is noticeably better than the cut-based model. It is interesting to note that both models did not lose more than 2% efficiency with higher p_T, while the cut-based model lost more than 6%.

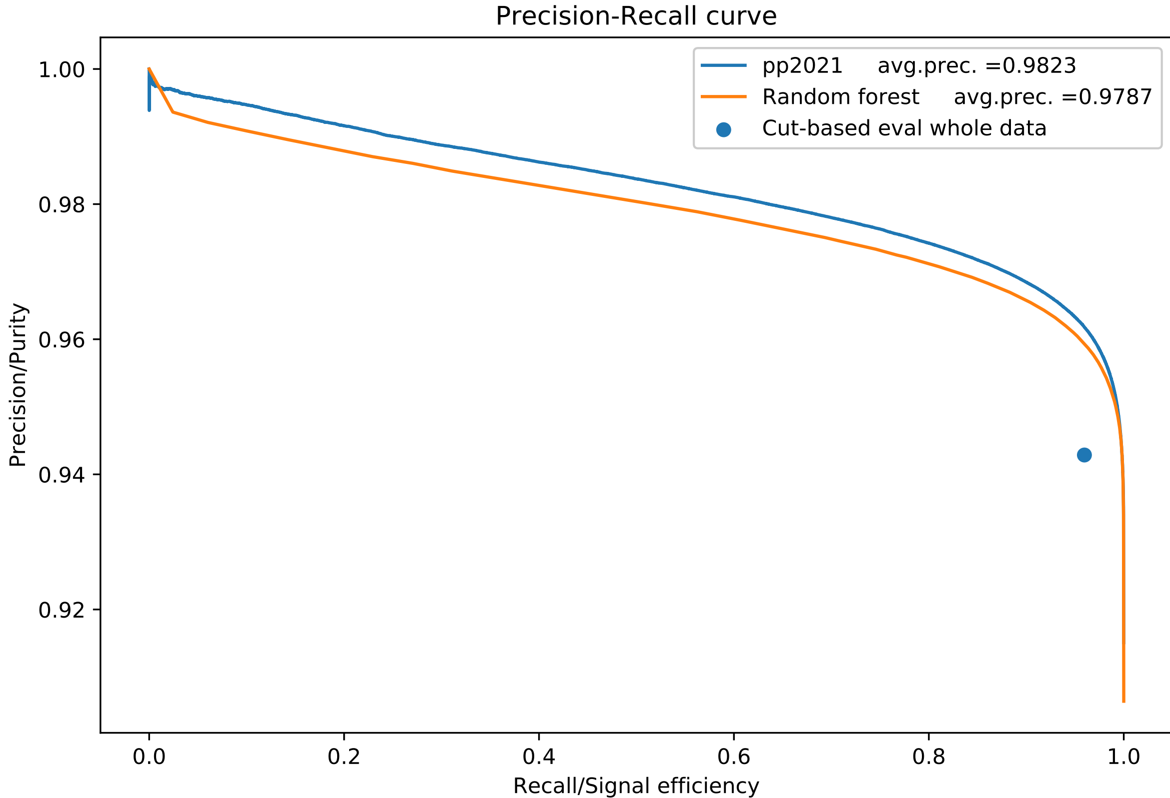


Figure 3.25: Purity-efficiency curves. (proton-proton collision system)

But when considering the purity-efficiency curve in figure 3.25, it is comprehensible. The proton-proton model *pp2021* reaches higher purities throughout the range of the signal efficiency. The XGBoost model has a significantly higher average precision score than the random forest model.

Lead-lead collision system

Similar to before, a random forest model was trained on the lead-lead dataset until it reached 300 members. The training was shorter with this dataset than with the larger proton-proton dataset and took 6 minutes and 30 seconds.

Figure A.9 in the appendix shows the adjustment of the purity to the cut-based model. The purity deviation of the models is noticeable at high transverse momentum, which could be lead to the small data subset size in the high transverse momentum bins.

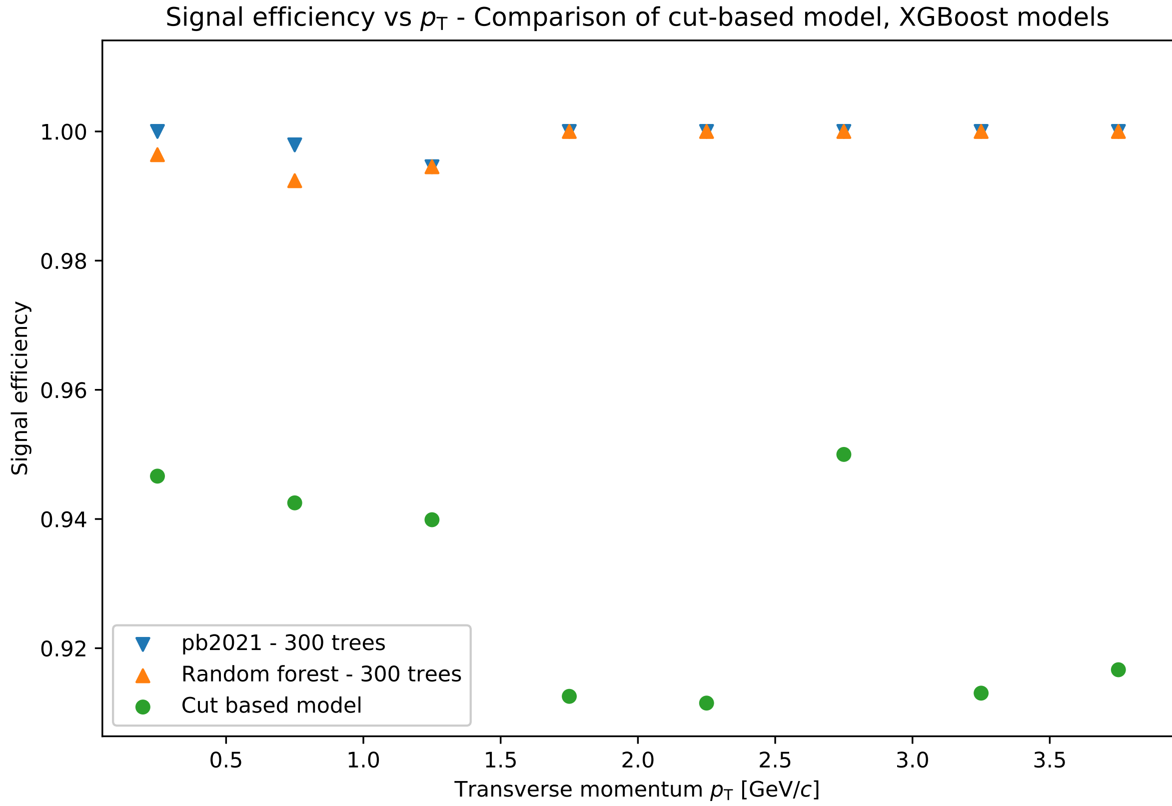


Figure 3.26: The signal efficiency of the models evaluated in each transverse momentum bin after the equalized purity. (lead-lead collision system)

As can be seen in figure A.9, both the random forest and the XGBoost model outperform the cut-based model by 5 to 9%. Except from the first two p_T-bins, the random forest model performed as well as the XGBoost model at nearly 100% efficiency. In the first two bins, the optimized XGBoost model outperforms the default random forest model.

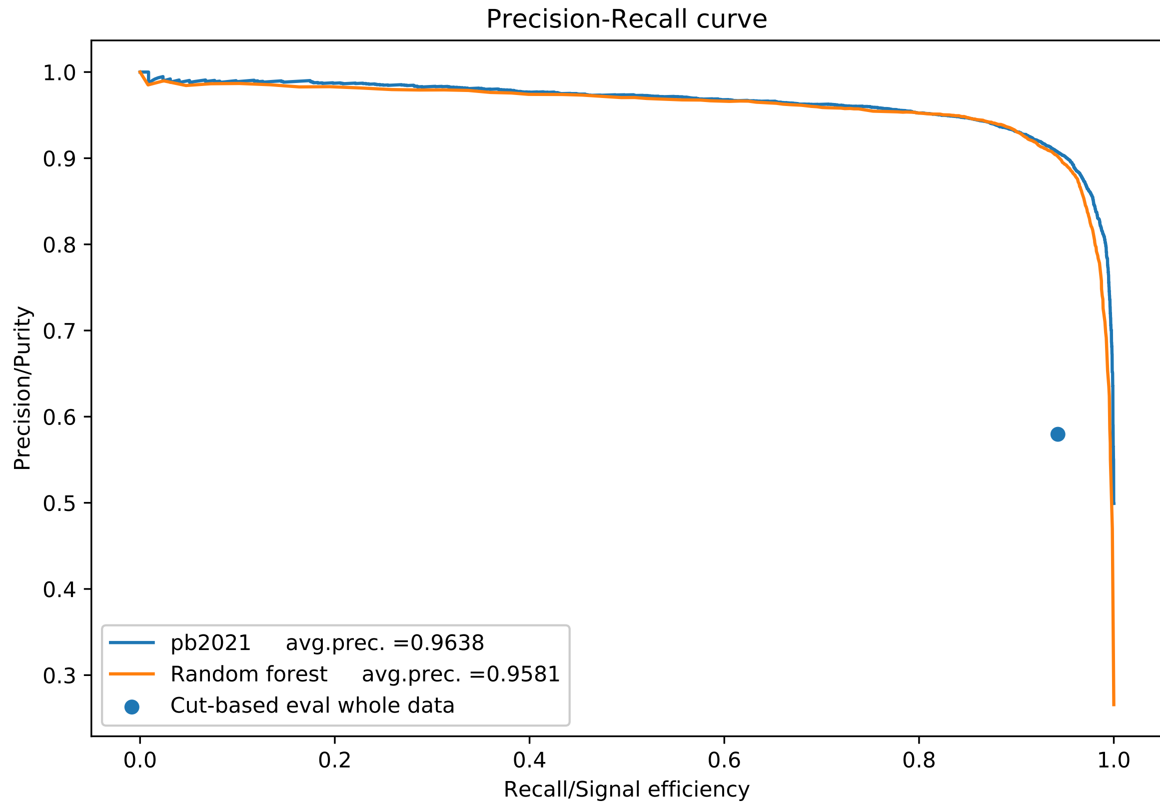


Figure 3.27: Precison Recall Curve.

The purity-efficiency curve in figure 3.27 shows that the difference in average precision score is a lot smaller than in the proton-proton collision system. But overall, the XGBoost model is significantly better.

3.4.5 Performance of the random forest model at high purities

Proton-proton collision system

Similar to section 3.4.2, the models will be compared at the purities 94, 96 and 98% (see figure A.10 in the appendix) in the proton-proton collision system. There were no significant deviations. Again, one begins with determining the probability threshold for the models in each transverse momentum bin to fit the goal purity.

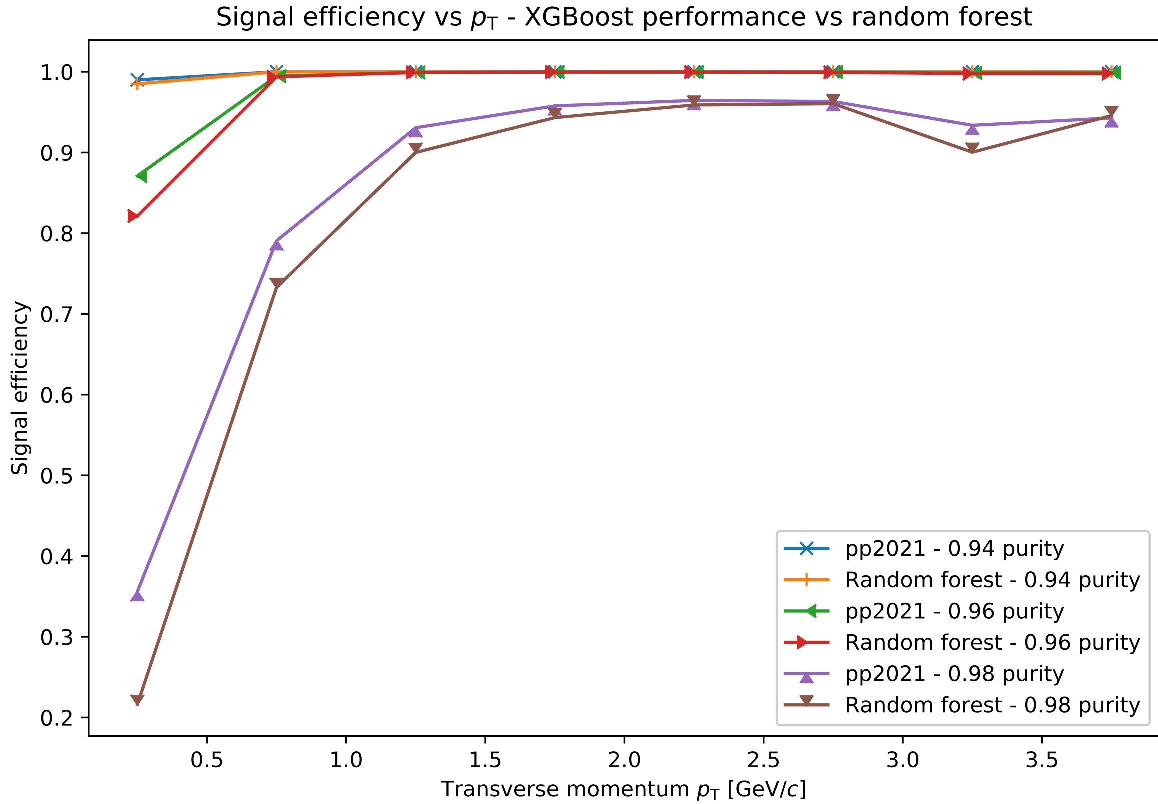


Figure 3.28: Signal efficiency of the XGBoost models at 94, 96 and 98% purity. (proton-proton collision system)

The signal efficiency at high purities is shown in figure 3.28. As one can see, the XGBoost model performs significantly better than the random forest model at high purities in the proton-proton collision system. Both models perform pretty well at 94 and 96% purity after the first p_T-bin, where they can keep up a signal efficiency very close to 100%. At 98% purity, the XGBoost model starts at about 35% efficiency, while the random forest model only achieves an efficiency slightly above 20%. Across the transverse momentum range between 0 and 4 GeV$/c$, the optimized XGBoost model achieves higher efficiency than the default random forest model. But after the second p_T bin around 1 GeV$/c$, both models can achieve an efficiency of around 90%. Both models are a good choice if only purity of 96% is required because the signal efficiency never drops below 80% signal efficiency.

Lead-lead collision system

In the lead-lead collision system, the models will be compared at 92, 95, and 98% purity (see figure) in the proton-proton collision system.
Because of the low abundance of primary photons in the lead-lead dataset, the goal purities deviated in some p_T-bins, especially at high p_T.

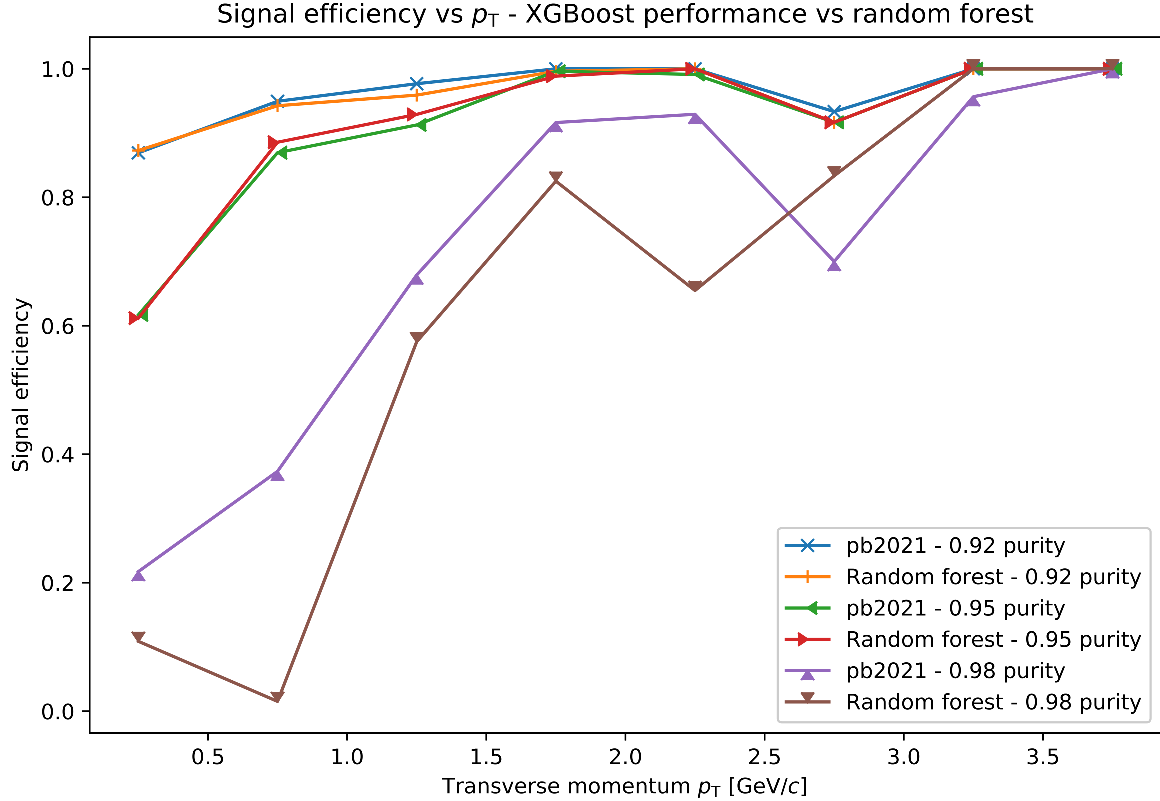


Figure 3.29: Signal efficiency of the XGBoost models at 92, 95 and 98% purity. (lead-lead collision system)

Figure 3.29 shows that the XGBoost tends to perform significantly better, especially at high purity. At a purity of 98%, the XGBoost model starts at around 20% signal efficiency. After a transverse momentum of approximately 1 GeV/c, the efficiency rises and stays above 60%. At a transverse momentum of 2.75 GeV/c, the XGBoost model has a large drop regarding efficiency. That is no surprise because the same validation dataset is used. At this point, the random forest outperforms the XGBoost model. But all in all, the optimized XGBoost and the default random forest model perform similarly well. Both perform significantly better than the cut-based model.

Chapter 4

Conclusion and discussion

In this thesis, different machine learning models were trained with two algorithms using Monte-Carlo simulations and compared to a cut-based model. By the use of Monte-Carlo simulations of lead-lead and proton-proton collisions in ALICE, it is possible to determine whether a data instance is a photon or not. Thus, one can compare the algorithms in terms of the efficiency and purity of predicted photons. Due to the high proportion of background in the lead-lead dataset, it is more challenging for the classifier to identify photons with high purity and efficiency. Therefore, the comparisons in the proton-proton collision system will not be mentioned hereafter. Among the different algorithms, the main focus is on the XGBoost algorithm. Some XGBoost models were trained with optimized hyperparameters to demonstrate the potential improvement. The XGBoost model without optimized hyperparameters achieved an average precision score of 0.9627, while the much simpler built random forest model achieved a score of 0.9581. The hyperparameter search improved the XGBoost model such that the average precision score increased to a value of 0.9638. It is, in any case, evident that even the simple random forest model achieves better results than the cut-based model. The precision-recall curves in the lead-lead collision system (see figures 3.23 and 3.27) indicate that machine learning models reach at least a 30% higher purity at the same signal efficiency. The efficiency of the trained models is not only higher but also more constant. In contrast, the cut-based model tends to lose efficiency with higher transverse momentum. Compared to the not optimized random forest model, the optimized XGBoost performed slightly better. But the difference between the machine learning models is negligible, while the difference between these machine learning models and the cut-based model is more significant. Meanwhile, the hyperparameter search only made a difference at the fourth decimal place. Possibly, the list of hyperparameters to be tested did not contain any good hyperparameters. Although the difference seems very small, one should keep in mind that it gets harder to increase the average precision score the closer one gets to an average precision score of 1. For instance, outliers are especially hard to identify. The difference among machine learning models might be more significant than the numbers indicate. In a nutshell, even with default hyperparameters or a simple algorithm such as the random forest, it is possible to train simple models with significantly better performance than the cut-based models. Statistical deviations of the targeted purity were very prominent in the lead-lead dataset, especially where the photon abundance was low. More precisely, the small size of the lead-lead dataset and the low quantity of data at high transverse momentum leads to the deviations. With larger datasets, one could train better models and counter these deviations. Another limiting factor is the quality of the Monte-Carlo simulation. Tiny high-dimensional feature deviations of the Monte-Carlo simulations from the experimental data could lead to a more significant bias in the decision process. Therefore, it could be interesting to evaluate these machine learning models on experimental datasets and compare the results with each other. It would also be interesting to see the difference between these results and those from simulation data.

Appendix A

Appendix

Python-code of the cut-based model

This code originates from Prof. Dr. Reygers [21].

```
def isInKinematicRange(row):

    #
    # cuts on electron/positron properties
    #

    # cuts and pseudo-rapidity range
    isElectronAndPostronInEtaRange = np.absolute(-np.log(np.tan(row.thetaElectron/2))
        ↪ ) < 0.9 \
        and np.absolute(-np.log(np.tan(row.thetaPositron/2))) < 0.9

    #
    # cuts on pair properties
    #
    isPhotonInEtaRange = np.absolute(-np.log(np.tan(row.theta/2))) < 0.9

    return isElectronAndPostronInEtaRange and isPhotonInEtaRange


def isPhotonCutBased(row):

    # Cuts in photon conversion analysis
    # https://github.com/alisw/AliPhysics/blob/master/PWGGA/GammaConvBase/
        ↪ AliConversionCuts.cxx

    # check whether the photon is measurable
    isInKinRange = isInKinematicRange(row)

    #
    # cuts on V0 track properties
    #

    # pt cuts
    isAbovePtThreshold = row.ptPositron > 0.05 and row.ptElectron > 0.05
```

```
# fraction TPC clusters
isOkFracClsTPC = row.fracClsTPCElectron > 0.6 and row.fracClsTPCPositron > 0.6

# dEdx is in electron band
isInElectronRange = row.nSigmaTPCeElectron > -3 and row.nSigmaTPCeElectron < 5 \
    and row.nSigmaTPCePositron > -3 and row.nSigmaTPCePositron < 5

# dEdx not in pion band
isNotInPionRangeElectron = True;
if row.ptElectron > 0.4 and row.ptElectron < 2:
    isNotInPionRangeElectron = row.nSigmaTPCpiElectron > 3.
if row.ptElectron >= 2:
    isNotInPionRangeElectron = row.nSigmaTPCpiElectron > 1.

isNotInPionRangePositron = True;
if (row.ptPositron > 0.4 and row.ptPositron < 2.):
    isNotInPionRangePositron = row.nSigmaTPCpiPositron > 3.
if (row.ptPositron >= 2.):
    isNotInPionRangePositron = row.nSigmaTPCpiPositron > 1.

# edited: "and" to "or"
isNotInPionRange = isNotInPionRangeElectron or isNotInPionRangePositron

# check TOF if available
isOkTofElectron = True
if row.nSigmaTOFElectron >= -19.:
    isOkTofElectron = np.absolute(row.nSigmaTOFElectron) < 5

isOkTofPositron = True
if row.nSigmaTOFPositron >= -19.:
    isOkTofPositron = np.absolute(row.nSigmaTOFPositron) < 5

isOkTof = isOkTofElectron and isOkTofPositron

# combination of cuts on electron and positron properties
isOkElectronPositronCuts = isAbovePtThreshold and isOkFracClsTPC and
    ↪ isInElectronRange and isNotInPionRange and isOkTof

#
# cuts on pair properties
#

# convers radius cut
isOkRcut = row.photonR > 5 # cm

# pointing angle
isOkPointingAngle = row.photonCosPoint > 0.85

# 2D Psi pair / chi^2 cut
PsiPairCut = 0.1
Chi2Cut = 30
isOkPsiPairVsChi2 = np.absolute(row.photonPsiPair) < PsiPairCut * (1. - row.
    ↪ chi2ndf/Chi2Cut)
```

```
    # photon qt cut (Armenteros-Podolanski)
    isOkPhotonQt = (row.photonQt/0.05)**2 + (row.photonAlpha/0.95)**2 < 1.

    # combination of cuts on pair properties
    isOkPairCuts = isOkPointingAngle and isOkPsiPairVsChi2 and isOkPhotonQt

    isPhoton = isInKinRange and isOkElectronPositronCuts and isOkPairCuts

    return isPhoton
```

XGBoost codes

Self-written class and functions.

```
# Necessary libraries-----------------------------------
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import xgboost as xgb
from xgboost import XGBClassifier
from sklearn import metrics
from sklearn.metrics import mean_squared_error as mse
import json
import pickle as pi
import time

# Own XGBoost model class--------------------------------


class SvenModel:
    """
    Memory efficient model
    """
    def __init__(
        self,
        modelname="new_model",
        load=False,
        loaddir="models/",
        nthread = 8,
        ):
        if load:
            mfile = loaddir+modelname+".model"
            pfile = loaddir+modelname+".txt"
            self.model = xgb.Booster({'nthread':nthread})
            self.model.load_model(mfile)
            try:
                f = open(pfile)
                self.model.feature_names = (f.readline()).split(",")
                self.name = f.readline()
                f.close()
            except:
                print("please set trained feature-indices with self.set_features()
                 ↪ functions")
```

```
        else:
            self.model = None
            self.params = param
        self.name = modelname


    def set_features(
        self,
        drop_features=["pt"],
        all_features=[
            'ptPositron',
            'thetaPositron',
            'dEdxPositronTPC',
            'tofPositron',
            'nSigmaTOFPositron',
            'fracClsTPCPositron',
            'clsITSPositron',
            'dEdxPositronITS',
            'clsTPCPositron',
            'nSigmaITSPositron',
            'ptElectron',
            'thetaElectron',
            'dEdxElectronTPC',
            'nSigmaTPCeElectron',
            'nSigmaTPCePositron',
            'tofElectron',
            'nSigmaTOFElectron',
            'fracClsTPCElectron',
            'clsITSElectron',
            'dEdxElectronITS',
            'clsTPCElectron',
            'nSigmaITSElectron',
            'nSigmaTPCpiPositron',
            'nSigmaTPCpiElectron',
            'photonQt',
            'photonAlpha',
            'photonPsiPair',
            'photonCosPoint',
            'photonInvMass',
            'photonX',
            'photonY',
            'photonZ',
            'photonPhi',
            'photonR',
            'pt',
            'theta',
            'chi2ndf'
        ]
    ):
        try:
            for i in drop_features:
                all_features.remove(i)
        except:
            pass
```

```
    self.model.feature_names = all_features


def save(self,savedir="models/",name=None):
    if name is None:
        mfile = savedir+self.name+".model"
        pfile = savedir+self.name+".txt"
    else:
        mfile = savedir+name+".model"
        pfile = savedir+self.name+".txt"
    #save model
    self.model.save_model(mfile)
    #save parameters
    f = open(pfile,"w+")
    f.write(",".join(self.model.feature_names))
    f.close()

def predict(self,features,thresholds=0.5):
    pred_score = self.model.predict(features)
    try: #different predictions with many different thresholds
        pred = np.array([(pred_score>=t).astype(int) for t in thresholds])
    except: #only one threshold
        pred = np.array(pred_score>=thresholds).astype(int)
    return pred

def predict_proba(self,val_dataframe):
    return self.model.predict(
        xgb.DMatrix(
            val_dataframe,#.iloc[:,:-1],
            #label=val_dataframe["kind"],
            feature_names=list(val_dataframe.columns)#[:-1]
        )
    )

def get_params(self):
    return self.model.get_params()

def plot_feature_importance(self,importancetype,save=False,figure_size=(12,5)):
    if importancetype not in ["weight","gain","cover","total_gain"]:
        print("Importance type not valid!\nPlease use one of...\n")
        print(["weight","gain","cover","total_gain"])
        raise
    fscore = self.model.get_score(importance_type=importancetype)
    values = np.fromiter(fscore.values(),dtype=float)
    sort = np.argsort(values)
    keys = np.array(list(fscore.keys()))

    fig = plt.figure(figsize=figure_size)
    fig.tight_layout()
    y_pos = np.arange(len(fscore))
    plt.barh(y_pos, values[sort], align='center')
    plt.yticks(y_pos, keys[sort])
    plt.xlabel(str('Feature importance: '+importancetype) )
    plt.title(str('Feature importance: '+importancetype+' of model: '+self.name))
```

```
        if save:
            plt.savefig(
                str("fimp_"+importancetype+"_"+self.name+".pdf"),
                dpi=200,
                format="pdf",
                bbox_inches='tight')
            plt.savefig(
                str("fimp_"+importancetype+"_"+self.name+".eps"),
                dpi=200,
                format="eps",
                bbox_inches='tight')
        plt.show()


# functions---------------------------------------

def load_partdata(csvfile,steps,part,use_features=[None],drop_features=["Unnamed: 0"
    ↪ ],numpy=False,binary=True,dmatrix=False):
    #count row number
    with open(csvfile) as fp:
        for (nrows, _) in enumerate(fp, 1):
            pass
    del fp
    if numpy:
        #load part of dataset and drop unwanted features
        if use_features[0] is None:
            df = (pd.read_csv(csvfile)\
                     .iloc[\
                          int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                 ).drop(columns=drop_features).to_numpy()
        else:
            df = (pd.read_csv(csvfile)[use_features]\
                     .iloc[\
                          int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                 ).to_numpy()
        #conversion of labels
        if binary:
            df[df[:,-1]!=0,-1]=1
            df[:,-1] = 1-df[:,-1]
    else:
        #load part of dataset and drop unwanted features
        if use_features[0] is None:
            df = (pd.read_csv(csvfile)\
                     .iloc[\
                          int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                 ).drop(columns=drop_features)
        else:
            df = (pd.read_csv(csvfile)[use_features]\
                     .iloc[\
                          int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                 )
        #conversion of labels
        if binary:
            df.loc[df["kind"]!=0, "kind"]=1
```

```
            df["kind"] = 1 - df["kind"]
        if dmatrix:
            df = xgb.DMatrix(df.iloc[:,:-1],label=df.iloc[:,-1],feature_names=list(df.
                ↪ columns)[:-1])
    return df

#
# Calculation functions
#
def calc_beff(predictions,labels):
    """
    calculate background efficiency
    """
    _, _, TN, _ = confusion_matrix(predictions,labels)

    beff = 1-TN/(len(labels)-np.sum(labels)) #background efficiency FPR

    return beff

def calc_purity(predictions, truth):
    """
    Uses confusion_matrix
    Returns PPV
    """
    TP, FP, TN, FN = confusion_matrix(predictions,truth)
    if len(predictions.shape) == 2:
        #different predictions with different thresholds
        tmp = np.sum((TP,FP),axis=0)
        TP[tmp==0.]=1.
        tmp[tmp==0.]=1.
    else:
        tmp = TP+FP
        if tmp == 0.:
            TP = 1.
            tmp = 1.
    pur = np.divide(TP,tmp)
    if type(pur) == np.ndarray:
        if len(pur) == 1:
            pur = pur[0]
    return pur


def check_kinrange(dataframe):
    """
    returns bool array
    """
    feature_names = list(dataframe.columns)
    features = dataframe.to_numpy()
    #maybe dangerous
    dataframe=None
    del dataframe
    idx_TE = np.argwhere(np.array(feature_names)=="thetaElectron")[:,0]
    idx_TP = np.argwhere(np.array(feature_names)=="thetaPositron")[:,0]
    idx_T = np.argwhere(np.array(feature_names)=="theta")[:,0]
```

```
    ep_eta_range = np.logical_and(
        np.absolute(-np.log(np.tan(features[:,idx_TE]/2))) < 0.9,
        np.absolute(-np.log(np.tan(features[:,idx_TP]/2))) < 0.9)
    photon_eta = np.absolute(-np.log(np.tan(features[:,idx_T]/2))) < 0.9
    return np.logical_and(
        ep_eta_range,
        photon_eta
    )

def confusion_matrix(predictions, truth):
    """
    this one only for binary
    0 is false, no photon
    1 is true, photon

    Input: prediction(s) and ground truth (as np.array)
    Output: True positive, false positive, true negative, false negative as numbers
        ↪ (1 threshold) or arrays (many thresholds)
    """
    TP, FP, TN, FN = [],[],[],[]
    if len(predictions.shape) == 2:
        for i in range(int(predictions.shape[0])):
            #print(i/len(threshold),"%")
            TP_ = np.sum(np.logical_and((predictions[i]==truth),(predictions[i]==1)))
            FP_ = np.sum(np.logical_and((predictions[i]!=truth),(predictions[i]==1)))
            TN_ = np.sum(np.logical_and((predictions[i]==truth),(predictions[i]==0)))
            FN_ = np.sum(np.logical_and((predictions[i]!=truth),(predictions[i]==0)))
            TP.append(TP_)
            FP.append(FP_)
            TN.append(TN_)
            FN.append(FN_)
        TP = np.array(TP)
        FP = np.array(FP)
        TN = np.array(TN)
        FN = np.array(FN)
    else:
        TP = np.sum(np.logical_and((predictions==truth),(predictions==1)))
        FP = np.sum(np.logical_and((predictions!=truth),(predictions==1)))
        TN = np.sum(np.logical_and((predictions==truth),(predictions==0)))
        FN = np.sum(np.logical_and((predictions!=truth),(predictions==0)))

    return TP, FP, TN, FN


#
# Comparison functions (cut-based vs xgboost)
#
def cut_purity_efficiency(tmp_df):
    val_dataframe = tmp_df.copy()
    val_dataframe['isPhotonCutBased'] = val_dataframe.apply(isPhotonCutBased, axis =
        ↪ 1)
    # photon purity without cuts
    n_photon_cand_no_cuts = len(val_dataframe)
    n_photons_no_cuts = np.sum(val_dataframe["kind"] == 1)
```

```
    purity_no_cuts = n_photons_no_cuts / n_photon_cand_no_cuts
    # efficiency and purity after cuts
    n_photon_cand = len(val_dataframe[val_dataframe["isPhotonCutBased"] == True]) #
        ↪ number photon candiates
    n_photons = len(val_dataframe[(val_dataframe["isPhotonCutBased"] == True) & (
        ↪ val_dataframe["kind"] == 1)])
    non_photons = len(val_dataframe[(val_dataframe["isPhotonCutBased"] == False) & (
        ↪ val_dataframe["kind"] == 0)])
    purity = n_photons / n_photon_cand
    eff = n_photons / n_photons_no_cuts

    non_photons_no_cuts = len(val_dataframe)-n_photons_no_cuts
    beff = 1-non_photons/ non_photons_no_cuts
    TPR = n_photons/(n_photons+len(val_dataframe[(val_dataframe["isPhotonCutBased"]
        ↪ == False) & (val_dataframe["kind"] == 1)]))
    val_dataframe = None
    del val_dataframe
    return purity_no_cuts, purity, eff, beff, TPR

def cut_by_pt(orig_df,pt_steps=8,pt_lim=(0,4)):
    df = orig_df.copy()
    df['isInKinematicRange'] = df.apply(isInKinematicRange, axis = 1)
    df['isPhotonCutBased'] = df.apply(isPhotonCutBased, axis = 1)
    #pt wise seperation
    df_pt_photon_cand_no_cuts = df[df["isInKinematicRange"] == True]["pt"]
    df_pt_photon_no_cuts = df[(df["isInKinematicRange"] == True) & (df["kind"] == 1)
        ↪ ]["pt"]
    df_pt_photon_cand = df[df["isPhotonCutBased"] == True]["pt"]
    df_pt_photons_after_cuts = df[(df["isPhotonCutBased"] == True) & (df["kind"] ==
        ↪ 1)]["pt"]
    #pt wise bin counting
    h_pt_photon_cand_no_cuts, edges_photon_cand_no_cuts = np.histogram(
        ↪ df_pt_photon_cand_no_cuts, bins=pt_steps, range=pt_lim)
    h_pt_photon_no_cuts, edges_true_photon_no_cuts = np.histogram(
        ↪ df_pt_photon_no_cuts, bins=pt_steps, range=pt_lim)
    h_pt_photon_cand, edges_photon_cand = np.histogram(df_pt_photon_cand, bins=
        ↪ pt_steps, range=pt_lim)
    h_pt_photons_after_cuts, edges_true_photons = np.histogram(
        ↪ df_pt_photons_after_cuts, bins=pt_steps, range=pt_lim)
    with np.errstate(divide='ignore', invalid='ignore'):
        purity_no_cuts_vs_pt = h_pt_photon_no_cuts / h_pt_photon_cand_no_cuts
        purity_vs_pt = h_pt_photons_after_cuts / h_pt_photon_cand
        eff_vs_pt = h_pt_photons_after_cuts / h_pt_photon_no_cuts
    return purity_vs_pt, eff_vs_pt, purity_no_cuts_vs_pt, edges_photon_cand

def evaluate_model(predprobs,labels):
    start = time.time()
    pur, eff, beff, _, _ = xgb_purity_efficiency(
        predictions = np.array([(predprobs>=thresholds).astype(int)])[0],
        labels = labels
    )
    end = time.time()
    print("Runtime cut-based evaluation: ",str(end-start)," s")
    print("Purity:\t\t",str(pur),"\nSignal efficiency\t",str(eff),"\nBackground
```

```
        ↪ efficiency\t",str(beff))

def find_pthreshold_beff(goal_beff,predprobs,labels,accuracy=5,max_iter=50,start_with
    ↪ =0.5):
    """
    Finds the right threshold to create similar background efficiencies
    Returns the probability threshold
    """
    tmp_th = start_with
    old_beff = 8888
    tmp_beff = calc_beff(
        predictions = np.array((predprobs>=tmp_th).astype(int)),
        labels = label
    )
    countdown=max_iter
    while round(tmp_beff,int(accuracy)) != round(goal_beff,int(accuracy)):
        print("Iteration ",str(max_iter-countdown),"\nBackround efficiency: ",str(
            ↪ tmp_beff),\
              "\nGoal: ",str(goal_beff),"Current threshold: ",str(tmp_th))
        if countdown < 1:
            break
        if (tmp_beff-goal_beff) > 0:
            tmp_th += 1./(2.**(max_iter-countdown+1))
        else:
            tmp_th -= 1./(2.**(max_iter-countdown+1))
        old_beff = tmp_beff
        tmp_beff = calc_beff(
            predictions = np.array((predprobs>=tmp_th).astype(int)),
            labels = labels
        )
        countdown-=1
    return tmp_th


def xgb_purity_efficiency(predictions, labels):
    """
    calculate purity and efficiency using predictions and labels
    """
    #isInKinRange = check_kinrange(features)[:,0]
    n_photons_no_cuts = np.sum(labels)
    TP, FP, TN, FN = confusion_matrix(predictions,labels)
    pur = TP/(TP+FP) #real photons in predicited photons aka. precision
    eff = TP/n_photons_no_cuts #signal efficiency aka. recall/TPR
    beff = 1-TN/(len(labels)-n_photons_no_cuts) #background efficiency FPR
    #print("Background efficiency check:\n",beff)
    #print(FP/(len(labels)-n_photons_no_cuts) )
    return pur, eff, beff, TP/(TP+FN), n_photons_no_cuts

def calc_pt_ints(val_dataframe,pt_lim=(0,4),pt_steps=8):
    pt_ints = [(pt_lim[0]+i*pt_lim[1]/pt_steps,pt_lim[0]+(i+1)*pt_lim[1]/pt_steps)
        ↪ for i in range(pt_steps)]
    #find indices of instances where pt value lies in the given pt range
    indx_by_pt = np.array(
        [(np.argwhere(
```

```
            np.logical_and((pt_ints[i][0] <= val_dataframe["pt"].values ),
                           (val_dataframe["pt"].values <= pt_ints[i][1]))
        )[:,0]).astype(int) for i in range(len(pt_ints))]
    )
    return pt_ints, indx_by_pt

def match_purity(goal_pur,predprobs,labels,accuracy=3,max_iter=100,start_with=0.5,
    ↪ print_messages=True):
    """
    Finds the right threshold to create similar purities
    Returns the probability threshold
    """
    tmp_th = start_with
    old_pur = 8888
    tmp_pur = calc_purity(
        predictions = np.array((predprobs>=tmp_th).astype(int)),
        truth = labels
    )
    countdown=max_iter
    while round(tmp_pur,int(accuracy)) != round(goal_pur,int(accuracy)):
        if print_messages:
            print("Iteration ",str(max_iter-countdown),"\nPurity: ",str(tmp_pur),\
                  "\nGoal: ",str(goal_pur),"Current threshold: ",str(tmp_th))
        if countdown < 1:
            break
        if (tmp_pur-goal_pur) > 0:
            tmp_th -= 1./(2.**(max_iter-countdown+1))
        else:
            tmp_th += 1./(2.**(max_iter-countdown+1))
        old_pur = tmp_pur
        tmp_pur = calc_purity(
            predictions = np.array((predprobs>=tmp_th).astype(int)),
            truth = labels
        )
        countdown-=1
    return tmp_th


def match_purities(predprobs,labels,indx_by_pt,purities):
    thresholds = []
    for i in range(len(indx_by_pt)):
        print("\n\nStep ",str(i),"/",str(len(indx_by_pt)))
        thresholds.append(
            match_purity(
                goal_pur = purities[i],
                predprobs=predprobs[indx_by_pt[i]],
                labels=labels[indx_by_pt[i]],
                accuracy = 3,
                max_iter = 200,
                print_messages=True#,
                #start_with = 0.3
            )
        )
    return thresholds
```

```
def eval_by_pt(predprobs,labels,cut_purities,pt_ints,indx_by_pt):
    purs = []
    effs = []
    beffs = []
    if cut_purities is not None:
        print("Matching purities: ")
        start = time.time()
        thresholds = match_purities(
            predprobs=predprobs,
            labels=labels,
            indx_by_pt=indx_by_pt,
            purities=cut_purities
        )
        end = time.time()
        print("Time passed: ",str(end-start),"s")
        print("\nNow evaluate XGBoost model by pt")
        start = time.time()
        for i in range(len(indx_by_pt)):
            pur, eff, beff, _, _ = xgb_purity_efficiency(
                predictions = np.array( predprobs[indx_by_pt[i]]>=thresholds[i] ).
                    ↪ astype(int),
                labels = labels[indx_by_pt[i]]
            )
            purs.append(pur)
            effs.append(eff)
            beffs.append(beff)
        end = time.time()
        print("Time passed: ",str(end-start),"s")
    else:
        print("\nEvaluate XGBoost model by pt")
        start = time.time()
        preds=np.array(predprobs>=0.5).astype(int)
        for i in range(len(indx_by_pt)):
            pur, eff, beff, _, _ = xgb_purity_efficiency(
                predictions = preds[indx_by_pt[i]],
                labels = labels[indx_by_pt[i]]
            )
            purs.append(pur)
            effs.append(eff)
            beffs.append(beff)
        end = time.time()
        print("Time passed: ",str(end-start),"s")
    return purs, effs, beffs

#
# Plot functions-----------------------------------
#
def plot_abundance(csvfile,figure_size=(12,6),save=False):
    kind_names = [
        "primary photons", #kind 0 photon
        "unspecified combinatorics", #kind 1
        "some mother, hadronic", #kind 2
        "pi0 Dalitz", #kind 3
```

```
        "eta Dalitz", #kind 4
        "secondary photons", #kind 5 photon
        "x", #kind 6
        "x", #kind 7
        "x", #kind 8
        "other",   #kind 9
        "electron combinatorial", #kind 10
        "pion combinatorial", #kind 11
        "pion proton combinatorics", #kind 12
        "pion electron combinatorics", #kind 13
        "kaon kaon combinatorics", #kind 14
        "direct electron combinatorial", #kind 15
        "electron protons combinatorics", #kind 16
        "electron kaon combinatorics", #kind 17
        "pion, kaon" #kind 18
    ]
    labels = pd.read_csv(csvfile).iloc[:,-1]
    plt.figure(figsize=figure_size)
    plt.subplot(211)
    counts, bins, patches = plt.hist(labels,18,color="#cc6900",edgecolor="black",
        ↪ linewidth=1)
    bin_centers = np.arange(19) + 0.5
    for count, x, kind in zip(counts, bin_centers, kind_names):
        # Label the raw counts
        plt.annotate(kind, xy=(x, 0), xycoords=('data', 'axes fraction'),
            xytext=(0, -40), textcoords='offset points', va='top', ha='center',
                ↪ rotation=270)
        # Label the percentages
        percent = '%0.1f%%' % (100. * float(count) / counts.sum())
        plt.annotate(percent, xy=(x, 0), xycoords=('data', 'axes fraction'),
            xytext=(0, -25), textcoords='offset points', va='top', ha='center')
    plt.yscale("log")
    plt.xticks(np.arange(19))
    plt.ylabel("counts")
    plt.title("Abundance of each photon kind")

    plt.subplot(212)
    plt.axis("Off")
    if save:
        plt.savefig("Abundances.pdf",dpi=200,format="pdf")
    plt.show()
```

XGBoost usage

Code snippets to show the usage.

```
# Necessary libraries-----------------------------------
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import xgboost as xgb
from xgboost import XGBClassifier
from sklearn import metrics
```

```
from sklearn.metrics import mean_squared_error as mse
import json
import pickle as pi
import time

# Own XGBoost model class-------------------------------


class SvenModel:
    """
    Memory efficient model
    """
    def __init__(
        self,
        modelname="new_model",
        load=False,
        loaddir="models/",
        nthread = 8,
        ):
        if load:
            mfile = loaddir+modelname+".model"
            pfile = loaddir+modelname+".txt"
            self.model = xgb.Booster({'nthread':nthread})
            self.model.load_model(mfile)
            try:
                f = open(pfile)
                self.model.feature_names = (f.readline()).split(",")
                self.name = f.readline()
                f.close()
            except:
                print("please set trained feature-indices with self.set_features()
                    ↪ functions")
        else:
            self.model = None
            self.params = param
        self.name = modelname


    def set_features(
        self,
        drop_features=["pt"],
        all_features=[
            'ptPositron',
            'thetaPositron',
            'dEdxPositronTPC',
            'tofPositron',
            'nSigmaTOFPositron',
            'fracClsTPCPositron',
            'clsITSPositron',
            'dEdxPositronITS',
            'clsTPCPositron',
            'nSigmaITSPositron',
            'ptElectron',
            'thetaElectron',
```

```
        'dEdxElectronTPC',
        'nSigmaTPCeElectron',
        'nSigmaTPCePositron',
        'tofElectron',
        'nSigmaTOFElectron',
        'fracClsTPCElectron',
        'clsITSElectron',
        'dEdxElectronITS',
        'clsTPCElectron',
        'nSigmaITSElectron',
        'nSigmaTPCpiPositron',
        'nSigmaTPCpiElectron',
        'photonQt',
        'photonAlpha',
        'photonPsiPair',
        'photonCosPoint',
        'photonInvMass',
        'photonX',
        'photonY',
        'photonZ',
        'photonPhi',
        'photonR',
        'pt',
        'theta',
        'chi2ndf'
    ]
):
    try:
        for i in drop_features:
            all_features.remove(i)
    except:
        pass
    self.model.feature_names = all_features


def save(self,savedir="models/",name=None):
    if name is None:
        mfile = savedir+self.name+".model"
        pfile = savedir+self.name+".txt"
    else:
        mfile = savedir+name+".model"
        pfile = savedir+self.name+".txt"
    #save model
    self.model.save_model(mfile)
    #save parameters
    f = open(pfile,"w+")
    f.write(",".join(self.model.feature_names))
    f.close()

def predict(self,features,thresholds=0.5):
    pred_score = self.model.predict(features)
    try: #different predictions with many different thresholds
        pred = np.array([(pred_score>=t).astype(int) for t in thresholds])
    except: #only one threshold
```

```
            pred = np.array(pred_score>=thresholds).astype(int)
        return pred

    def predict_proba(self,val_dataframe):
        return self.model.predict(
            xgb.DMatrix(
                val_dataframe,#.iloc[:,:-1],
                #label=val_dataframe["kind"],
                feature_names=list(val_dataframe.columns)#[:-1]
            )
        )

    def get_params(self):
        return self.model.get_params()

    def plot_feature_importance(self,importancetype,save=False,figure_size=(12,5)):
        if importancetype not in ["weight","gain","cover","total_gain"]:
            print("Importance type not valid!\nPlease use one of...\n")
            print(["weight","gain","cover","total_gain"])
            raise
        fscore = self.model.get_score(importance_type=importancetype)
        values = np.fromiter(fscore.values(),dtype=float)
        sort = np.argsort(values)
        keys = np.array(list(fscore.keys()))

        fig = plt.figure(figsize=figure_size)
        fig.tight_layout()
        y_pos = np.arange(len(fscore))
        plt.barh(y_pos, values[sort], align='center')
        plt.yticks(y_pos, keys[sort])
        plt.xlabel(str('Feature importance: '+importancetype) )
        plt.title(str('Feature importance: '+importancetype+' of model: '+self.name))
        if save:
            plt.savefig(
                str("fimp_"+importancetype+"_"+self.name+".pdf"),
                dpi=200,
                format="pdf",
                bbox_inches='tight')
            plt.savefig(
                str("fimp_"+importancetype+"_"+self.name+".eps"),
                dpi=200,
                format="eps",
                bbox_inches='tight')
        plt.show()


# functions-----------------------------------------

def load_partdata(csvfile,steps,part,use_features=[None],drop_features=["Unnamed: 0"
    ↪ ],numpy=False,binary=True,dmatrix=False):
    #count row number
    with open(csvfile) as fp:
        for (nrows, _) in enumerate(fp, 1):
            pass
```

```
    del fp
    if numpy:
        #load part of dataset and drop unwanted features
        if use_features[0] is None:
            df = (pd.read_csv(csvfile)\
                    .iloc[\
                        int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                ).drop(columns=drop_features).to_numpy()
        else:
            df = (pd.read_csv(csvfile)[use_features]\
                    .iloc[\
                        int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                ).to_numpy()
        #conversion of labels
        if binary:
            df[df[:,-1]!=0,-1]=1
            df[:,-1] = 1-df[:,-1]
    else:
        #load part of dataset and drop unwanted features
        if use_features[0] is None:
            df = (pd.read_csv(csvfile)\
                    .iloc[\
                        int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                ).drop(columns=drop_features)
        else:
            df = (pd.read_csv(csvfile)[use_features]\
                    .iloc[\
                        int(part*(nrows/steps)):int((part+1)*(nrows/steps))]
                )
        #conversion of labels
        if binary:
            df.loc[df["kind"]!=0, "kind"]=1
            df["kind"] = 1 - df["kind"]
        if dmatrix:
            df = xgb.DMatrix(df.iloc[:,:-1],label=df.iloc[:,-1],feature_names=list(df.
                ↪ columns)[:-1])
    return df

#
# Calculation functions
#
def calc_beff(predictions,labels):
    """
    calculate background efficiency
    """
    _, _, TN, _ = confusion_matrix(predictions,labels)

    beff = 1-TN/(len(labels)-np.sum(labels)) #background efficiency FPR

    return beff

def calc_purity(predictions, truth):
    """
    Uses confusion_matrix
```

```
    Returns PPV
    """
    TP, FP, TN, FN = confusion_matrix(predictions,truth)
    if len(predictions.shape) == 2:
        #different predictions with different thresholds
        tmp = np.sum((TP,FP),axis=0)
        TP[tmp==0.]=1.
        tmp[tmp==0.]=1.
    else:
        tmp = TP+FP
        if tmp == 0.:
            TP = 1.
            tmp = 1.
    pur = np.divide(TP,tmp)
    if type(pur) == np.ndarray:
        if len(pur) == 1:
            pur = pur[0]
    return pur


def check_kinrange(dataframe):
    """
    returns bool array
    """
    feature_names = list(dataframe.columns)
    features = dataframe.to_numpy()
    #maybe dangerous
    dataframe=None
    del dataframe
    idx_TE = np.argwhere(np.array(feature_names)=="thetaElectron")[:,0]
    idx_TP = np.argwhere(np.array(feature_names)=="thetaPositron")[:,0]
    idx_T = np.argwhere(np.array(feature_names)=="theta")[:,0]
    ep_eta_range = np.logical_and(
        np.absolute(-np.log(np.tan(features[:,idx_TE]/2))) < 0.9,
        np.absolute(-np.log(np.tan(features[:,idx_TP]/2))) < 0.9)
    photon_eta = np.absolute(-np.log(np.tan(features[:,idx_T]/2))) < 0.9
    return np.logical_and(
        ep_eta_range,
        photon_eta
    )

def confusion_matrix(predictions, truth):
    """
    this one only for binary
    0 is false, no photon
    1 is true, photon

    Input: prediction(s) and ground truth (as np.array)
    Output: True positive, false positive, true negative, false negative as numbers
        ↪ (1 threshold) or arrays (many thresholds)
    """
    TP, FP, TN, FN = [],[],[],[]
    if len(predictions.shape) == 2:
        for i in range(int(predictions.shape[0])):
```

```
            #print(i/len(threshold),"%")
            TP_ = np.sum(np.logical_and((predictions[i]==truth),(predictions[i]==1)))
            FP_ = np.sum(np.logical_and((predictions[i]!=truth),(predictions[i]==1)))
            TN_ = np.sum(np.logical_and((predictions[i]==truth),(predictions[i]==0)))
            FN_ = np.sum(np.logical_and((predictions[i]!=truth),(predictions[i]==0)))
            TP.append(TP_)
            FP.append(FP_)
            TN.append(TN_)
            FN.append(FN_)
        TP = np.array(TP)
        FP = np.array(FP)
        TN = np.array(TN)
        FN = np.array(FN)
    else:
        TP = np.sum(np.logical_and((predictions==truth),(predictions==1)))
        FP = np.sum(np.logical_and((predictions!=truth),(predictions==1)))
        TN = np.sum(np.logical_and((predictions==truth),(predictions==0)))
        FN = np.sum(np.logical_and((predictions!=truth),(predictions==0)))

    return TP, FP, TN, FN


#
# Comparison functions (cut-based vs xgboost)
#
def cut_purity_efficiency(tmp_df):
    val_dataframe = tmp_df.copy()
    val_dataframe['isPhotonCutBased'] = val_dataframe.apply(isPhotonCutBased, axis =
        ↪ 1)
    # photon purity without cuts
    n_photon_cand_no_cuts = len(val_dataframe)
    n_photons_no_cuts = np.sum(val_dataframe["kind"] == 1)
    purity_no_cuts = n_photons_no_cuts / n_photon_cand_no_cuts
    # efficiency and purity after cuts
    n_photon_cand = len(val_dataframe[val_dataframe["isPhotonCutBased"] == True]) #
        ↪ number photon candiates
    n_photons = len(val_dataframe[(val_dataframe["isPhotonCutBased"] == True) & (
        ↪ val_dataframe["kind"] == 1)])
    non_photons = len(val_dataframe[(val_dataframe["isPhotonCutBased"] == False) & (
        ↪ val_dataframe["kind"] == 0)])
    purity = n_photons / n_photon_cand
    eff = n_photons / n_photons_no_cuts

    non_photons_no_cuts = len(val_dataframe)-n_photons_no_cuts
    beff = 1-non_photons/ non_photons_no_cuts
    TPR = n_photons/(n_photons+len(val_dataframe[(val_dataframe["isPhotonCutBased"]
        ↪ == False) & (val_dataframe["kind"] == 1)]))
    val_dataframe = None
    del val_dataframe
    return purity_no_cuts, purity, eff, beff, TPR

def cut_by_pt(orig_df,pt_steps=8,pt_lim=(0,4)):
    df = orig_df.copy()
    df['isInKinematicRange'] = df.apply(isInKinematicRange, axis = 1)
```

```
    df['isPhotonCutBased'] = df.apply(isPhotonCutBased, axis = 1)
    #pt wise seperation
    df_pt_photon_cand_no_cuts = df[df["isInKinematicRange"] == True]["pt"]
    df_pt_photon_no_cuts = df[(df["isInKinematicRange"] == True) & (df["kind"] == 1)
        ↪ ]["pt"]
    df_pt_photon_cand = df[df["isPhotonCutBased"] == True]["pt"]
    df_pt_photons_after_cuts = df[(df["isPhotonCutBased"] == True) & (df["kind"] ==
        ↪ 1)]["pt"]
    #pt wise bin counting
    h_pt_photon_cand_no_cuts, edges_photon_cand_no_cuts = np.histogram(
        ↪ df_pt_photon_cand_no_cuts, bins=pt_steps, range=pt_lim)
    h_pt_photon_no_cuts, edges_true_photon_no_cuts = np.histogram(
        ↪ df_pt_photon_no_cuts, bins=pt_steps, range=pt_lim)
    h_pt_photon_cand, edges_photon_cand = np.histogram(df_pt_photon_cand, bins=
        ↪ pt_steps, range=pt_lim)
    h_pt_photons_after_cuts, edges_true_photons = np.histogram(
        ↪ df_pt_photons_after_cuts, bins=pt_steps, range=pt_lim)
    with np.errstate(divide='ignore', invalid='ignore'):
        purity_no_cuts_vs_pt = h_pt_photon_no_cuts / h_pt_photon_cand_no_cuts
        purity_vs_pt = h_pt_photons_after_cuts / h_pt_photon_cand
        eff_vs_pt = h_pt_photons_after_cuts / h_pt_photon_no_cuts
    return purity_vs_pt, eff_vs_pt, purity_no_cuts_vs_pt, edges_photon_cand

def evaluate_model(predprobs,labels):
    start = time.time()
    pur, eff, beff, _, _ = xgb_purity_efficiency(
        predictions = np.array([(predprobs>=thresholds).astype(int)])[0],
        labels = labels
    )
    end = time.time()
    print("Runtime cut-based evaluation: ",str(end-start)," s")
    print("Purity:\t\t",str(pur),"\nSignal efficiency\t",str(eff),"\nBackground
        ↪ efficiency\t",str(beff))

def find_pthreshold_beff(goal_beff,predprobs,labels,accuracy=5,max_iter=50,start_with
    ↪ =0.5):
    """
    Finds the right threshold to create similar background efficiencies
    Returns the probability threshold
    """
    tmp_th = start_with
    old_beff = 8888
    tmp_beff = calc_beff(
        predictions = np.array((predprobs>=tmp_th).astype(int)),
        labels = label
    )
    countdown=max_iter
    while round(tmp_beff,int(accuracy)) != round(goal_beff,int(accuracy)):
        print("Iteration ",str(max_iter-countdown),"\nBackround efficiency: ",str(
            ↪ tmp_beff),\
             "\nGoal: ",str(goal_beff),"Current threshold: ",str(tmp_th))
        if countdown < 1:
            break
        if (tmp_beff-goal_beff) > 0:
```

```
            tmp_th += 1./(2.**(max_iter-countdown+1))
        else:
            tmp_th -= 1./(2.**(max_iter-countdown+1))
        old_beff = tmp_beff
        tmp_beff = calc_beff(
            predictions = np.array((predprobs>=tmp_th).astype(int)),
            labels = labels
        )
        countdown-=1
    return tmp_th


def xgb_purity_efficiency(predictions, labels):
    """
    calculate purity and efficiency using predictions and labels
    """
    #isInKinRange = check_kinrange(features)[:,0]
    n_photons_no_cuts = np.sum(labels)
    TP, FP, TN, FN = confusion_matrix(predictions,labels)
    pur = TP/(TP+FP) #real photons in predicited photons aka. precision
    eff = TP/n_photons_no_cuts #signal efficiency aka. recall/TPR
    beff = 1-TN/(len(labels)-n_photons_no_cuts) #background efficiency FPR
    #print("Background efficiency check:\n",beff)
    #print(FP/(len(labels)-n_photons_no_cuts) )
    return pur, eff, beff, TP/(TP+FN), n_photons_no_cuts

def calc_pt_ints(val_dataframe,pt_lim=(0,4),pt_steps=8):
    pt_ints = [(pt_lim[0]+i*pt_lim[1]/pt_steps,pt_lim[0]+(i+1)*pt_lim[1]/pt_steps)
        ↪ for i in range(pt_steps)]
    #find indices of instances where pt value lies in the given pt range
    indx_by_pt = np.array(
        [(np.argwhere(
            np.logical_and((pt_ints[i][0] <= val_dataframe["pt"].values ),
                        (val_dataframe["pt"].values <= pt_ints[i][1]))
        )[:,0]).astype(int) for i in range(len(pt_ints))]
    )
    return pt_ints, indx_by_pt

def match_purity(goal_pur,predprobs,labels,accuracy=3,max_iter=100,start_with=0.5,
    ↪ print_messages=True):
    """
    Finds the right threshold to create similar purities
    Returns the probability threshold
    """
    tmp_th = start_with
    old_pur = 8888
    tmp_pur = calc_purity(
        predictions = np.array((predprobs>=tmp_th).astype(int)),
        truth = labels
    )
    countdown=max_iter
    while round(tmp_pur,int(accuracy)) != round(goal_pur,int(accuracy)):
        if print_messages:
            print("Iteration ",str(max_iter-countdown),"\nPurity: ",str(tmp_pur),\
```

```
            "\nGoal: ",str(goal_pur),"Current threshold: ",str(tmp_th))
        if countdown < 1:
            break
        if (tmp_pur-goal_pur) > 0:
            tmp_th -= 1./(2.**(max_iter-countdown+1))
        else:
            tmp_th += 1./(2.**(max_iter-countdown+1))
        old_pur = tmp_pur
        tmp_pur = calc_purity(
            predictions = np.array((predprobs>=tmp_th).astype(int)),
            truth = labels
        )
        countdown-=1
    return tmp_th


def match_purities(predprobs,labels,indx_by_pt,purities):
    thresholds = []
    for i in range(len(indx_by_pt)):
        print("\n\nStep ",str(i),"/",str(len(indx_by_pt)))
        thresholds.append(
            match_purity(
                goal_pur = purities[i],
                predprobs=predprobs[indx_by_pt[i]],
                labels=labels[indx_by_pt[i]],
                accuracy = 3,
                max_iter = 200,
                print_messages=True#,
                #start_with = 0.3
            )
        )
    return thresholds

def eval_by_pt(predprobs,labels,cut_purities,pt_ints,indx_by_pt):
    purs = []
    effs = []
    beffs = []
    if cut_purities is not None:
        print("Matching purities: ")
        start = time.time()
        thresholds = match_purities(
            predprobs=predprobs,
            labels=labels,
            indx_by_pt=indx_by_pt,
            purities=cut_purities
        )
        end = time.time()
        print("Time passed: ",str(end-start),"s")
        print("\nNow evaluate XGBoost model by pt")
        start = time.time()
        for i in range(len(indx_by_pt)):
            pur, eff, beff, _, _ = xgb_purity_efficiency(
                predictions = np.array( predprobs[indx_by_pt[i]]>=thresholds[i] ).
                    ↪ astype(int),
```

```
            labels = labels[indx_by_pt[i]]
        )
        purs.append(pur)
        effs.append(eff)
        beffs.append(beff)
    end = time.time()
    print("Time passed: ",str(end-start),"s")
else:
    print("\nEvaluate XGBoost model by pt")
    start = time.time()
    preds=np.array(predprobs>=0.5).astype(int)
    for i in range(len(indx_by_pt)):
        pur, eff, beff, _, _ = xgb_purity_efficiency(
            predictions = preds[indx_by_pt[i]],
            labels = labels[indx_by_pt[i]]
        )
        purs.append(pur)
        effs.append(eff)
        beffs.append(beff)
    end = time.time()
    print("Time passed: ",str(end-start),"s")
return purs, effs, beffs

#
# Plot functions-----------------------------------
#
def plot_abundance(csvfile,figure_size=(12,6),save=False):
    kind_names = [
        "primary photons", #kind 0 photon
        "unspecified combinatorics", #kind 1
        "some mother, hadronic", #kind 2
        "pi0 Dalitz", #kind 3
        "eta Dalitz", #kind 4
        "secondary photons", #kind 5 photon
        "x", #kind 6
        "x", #kind 7
        "x", #kind 8
        "other",   #kind 9
        "electron combinatorial", #kind 10
        "pion combinatorial", #kind 11
        "pion proton combinatorics", #kind 12
        "pion electron combinatorics", #kind 13
        "kaon kaon combinatorics", #kind 14
        "direct electron combinatorial", #kind 15
        "electron protons combinatorics", #kind 16
        "electron kaon combinatorics", #kind 17
        "pion, kaon" #kind 18
    ]
    labels = pd.read_csv(csvfile).iloc[:,-1]
    plt.figure(figsize=figure_size)
    plt.subplot(211)
    counts, bins, patches = plt.hist(labels,18,color="#cc6900",edgecolor="black",
        ↪ linewidth=1)
    bin_centers = np.arange(19) + 0.5
```

```
for count, x, kind in zip(counts, bin_centers, kind_names):
    # Label the raw counts
    plt.annotate(kind, xy=(x, 0), xycoords=('data', 'axes fraction'),
        xytext=(0, -40), textcoords='offset points', va='top', ha='center',
            ↪ rotation=270)
    # Label the percentages
    percent = '%0.1f%%' % (100. * float(count) / counts.sum())
    plt.annotate(percent, xy=(x, 0), xycoords=('data', 'axes fraction'),
        xytext=(0, -25), textcoords='offset points', va='top', ha='center')
plt.yscale("log")
plt.xticks(np.arange(19))
plt.ylabel("counts")
plt.title("Abundance of each photon kind")

plt.subplot(212)
plt.axis("Off")
if save:
    plt.savefig("Abundances.pdf",dpi=200,format="pdf")
plt.show()
```

Feature importance

Because a few features share a large proportion of the importance, it stands to reason that plotting them against each other should extend the comprehension of the models. Two- and three-dimensional features are shown in the subsequent figures. If these features carry valuable information to identify photons, the plots should show distinct clusters of photons and background. These two and three-dimensional plots only serve as rough insights because the classification happens in the high dimensional space. Moreover, the background data points in the plots are transparent to a certain amount. That is necessary to make the distribution of both photons and background data clearer. These plots should point out the different distributions of photon and background data.
Figure A.1 shows four different feature combinations. Each plot was done with the same 500000 data instances of the proton-proton collision system. Although the photon and background data overlap in the feature plots, one can notice different densities in different regions.

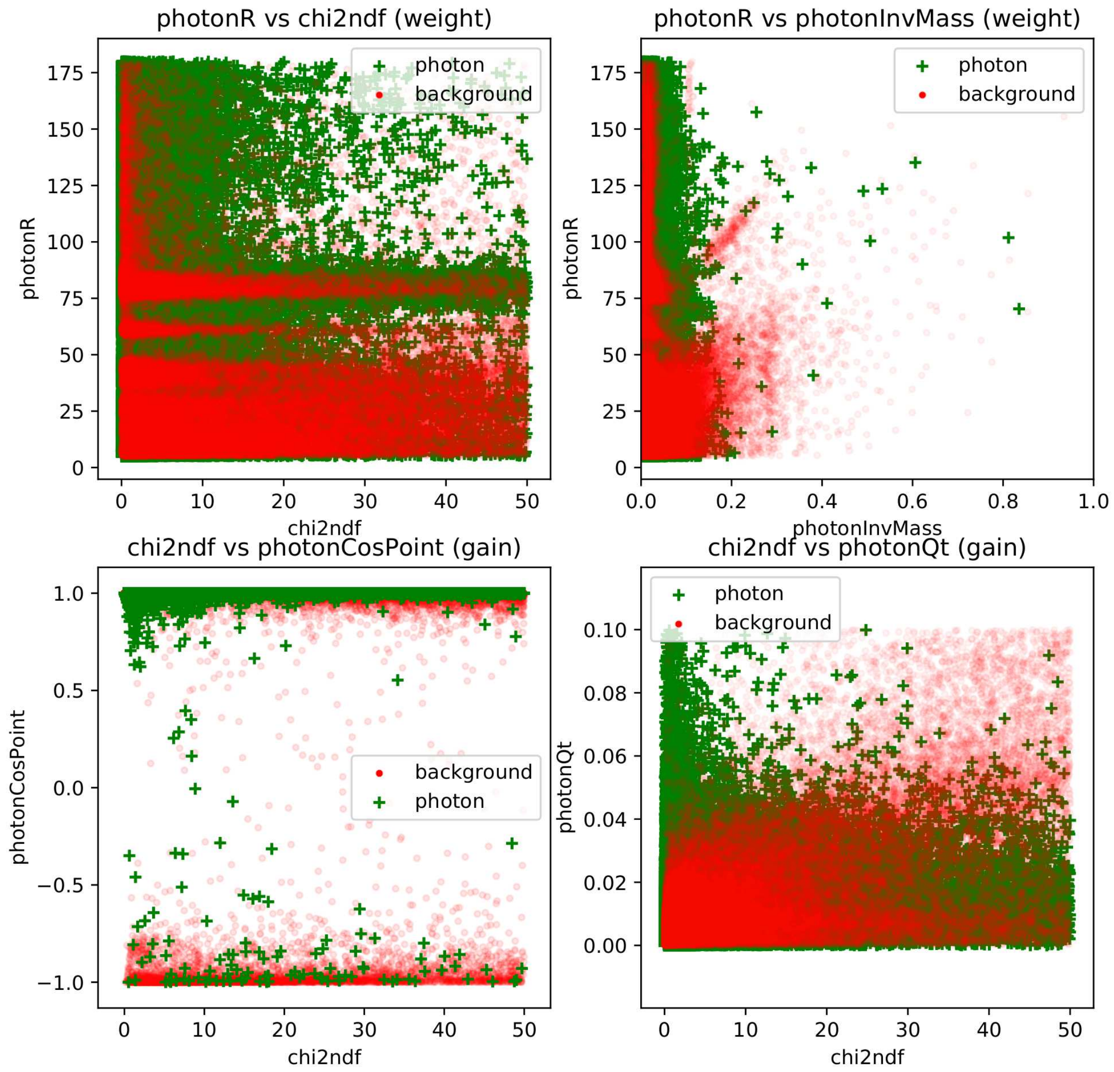
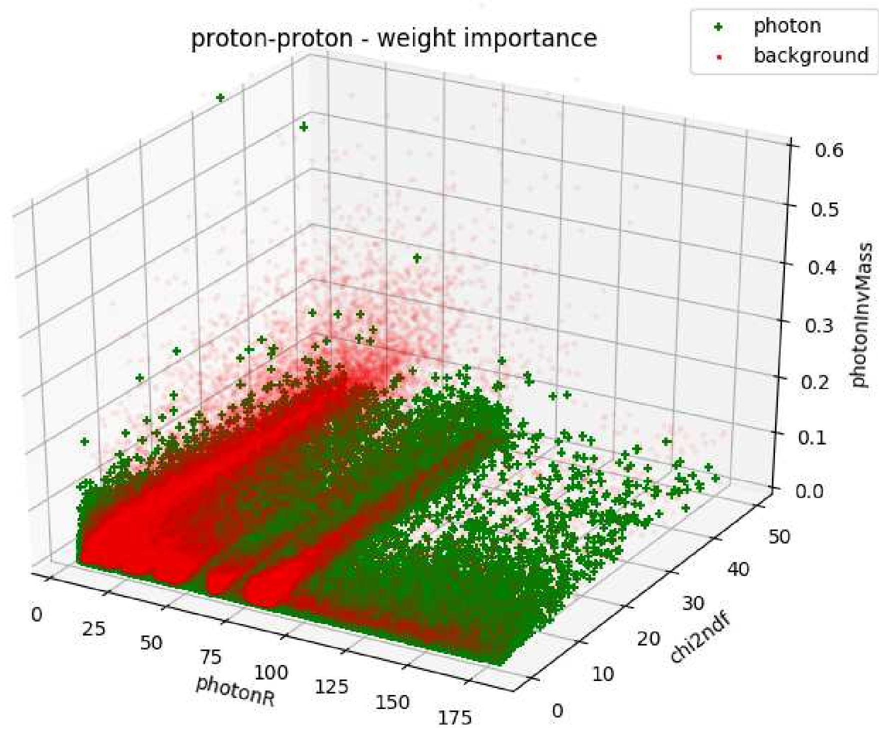
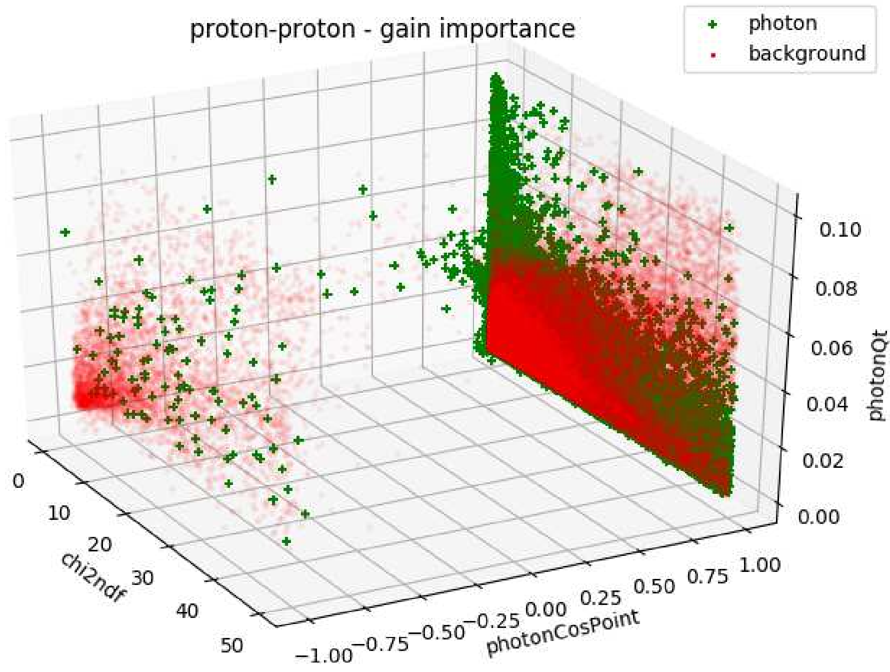


Figure A.1: The three most important features of the proton-proton model *pp2021* plotted in 2D.



(a) Weight.



(b) Gain.

Figure A.2: The three most important features of the proton-proton model *pp2021* plotted in 3D.

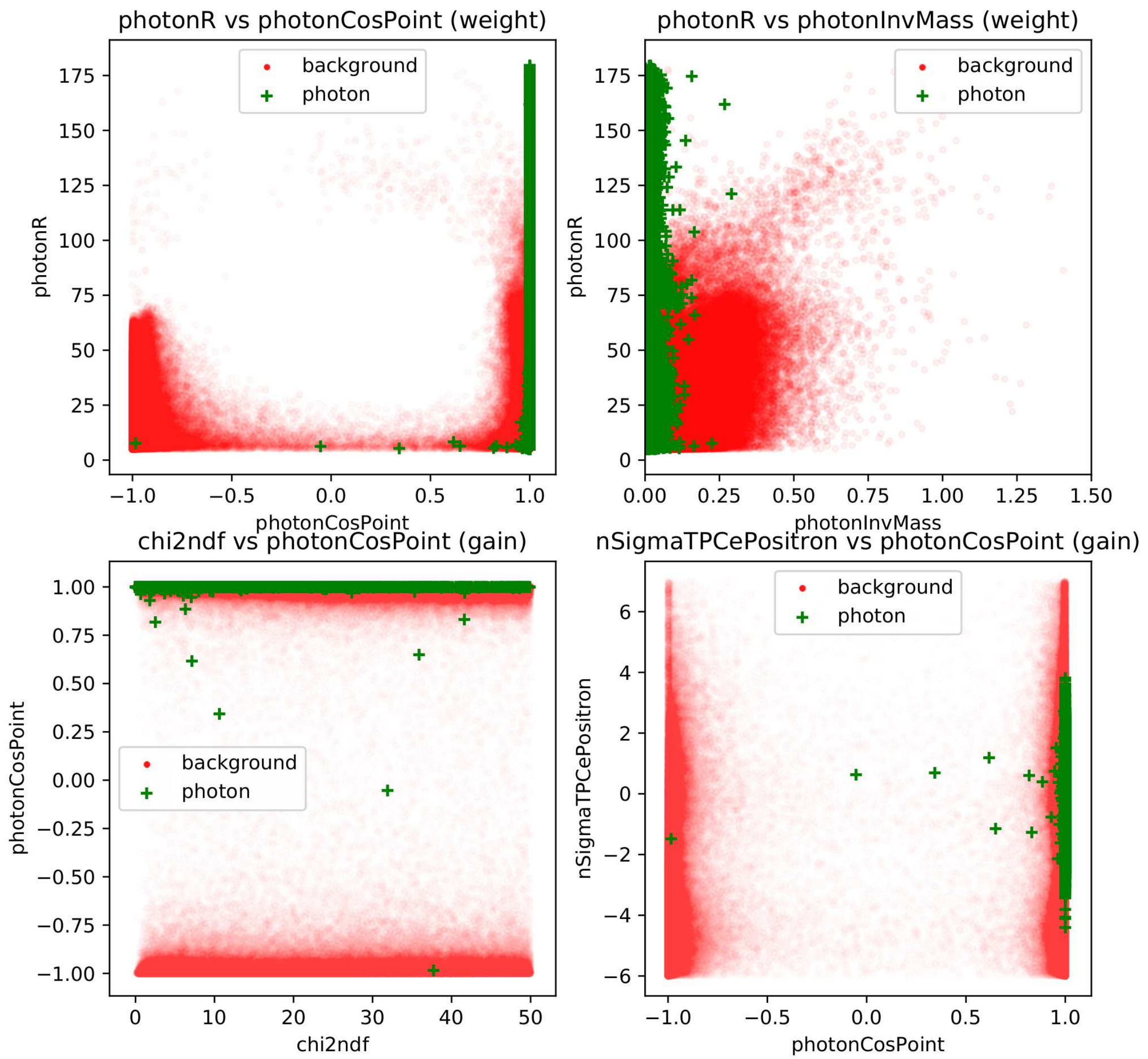
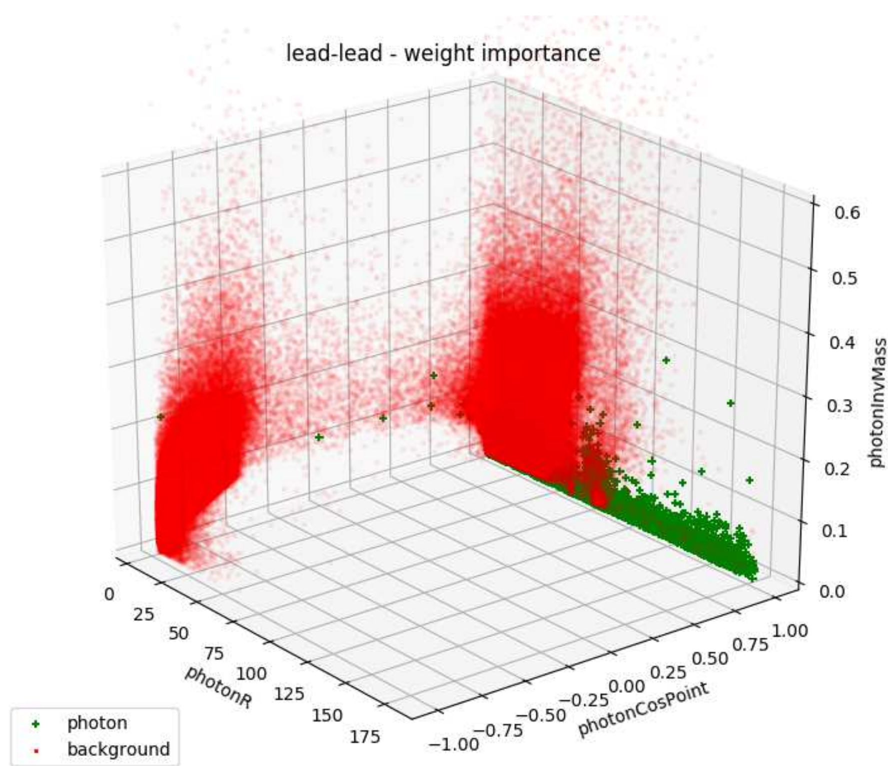
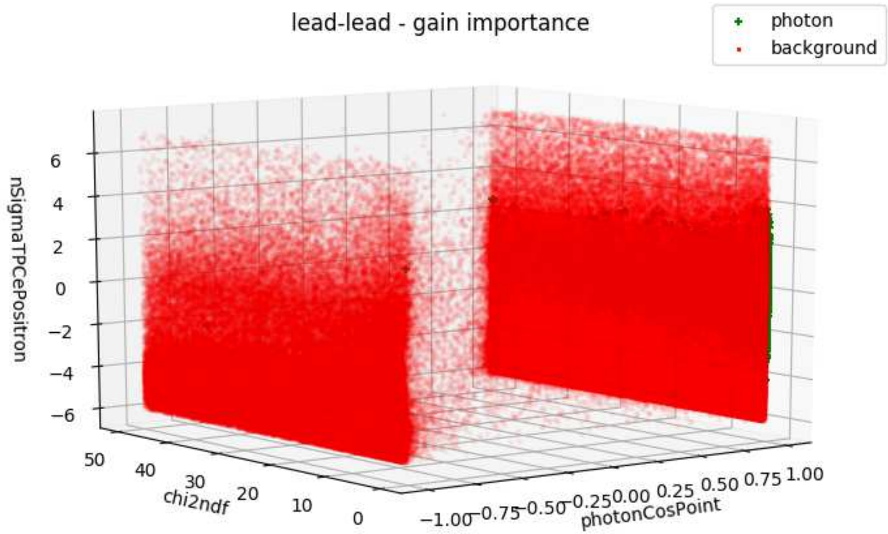


Figure A.3: The three most important features of the lead-lead model *pb2021* plotted in 2D.



(a) Weight.



(b) Gain.

Figure A.4: The three most important features of the lead-lead model *pb2021* plotted in 3D.

XGBoost evaluation

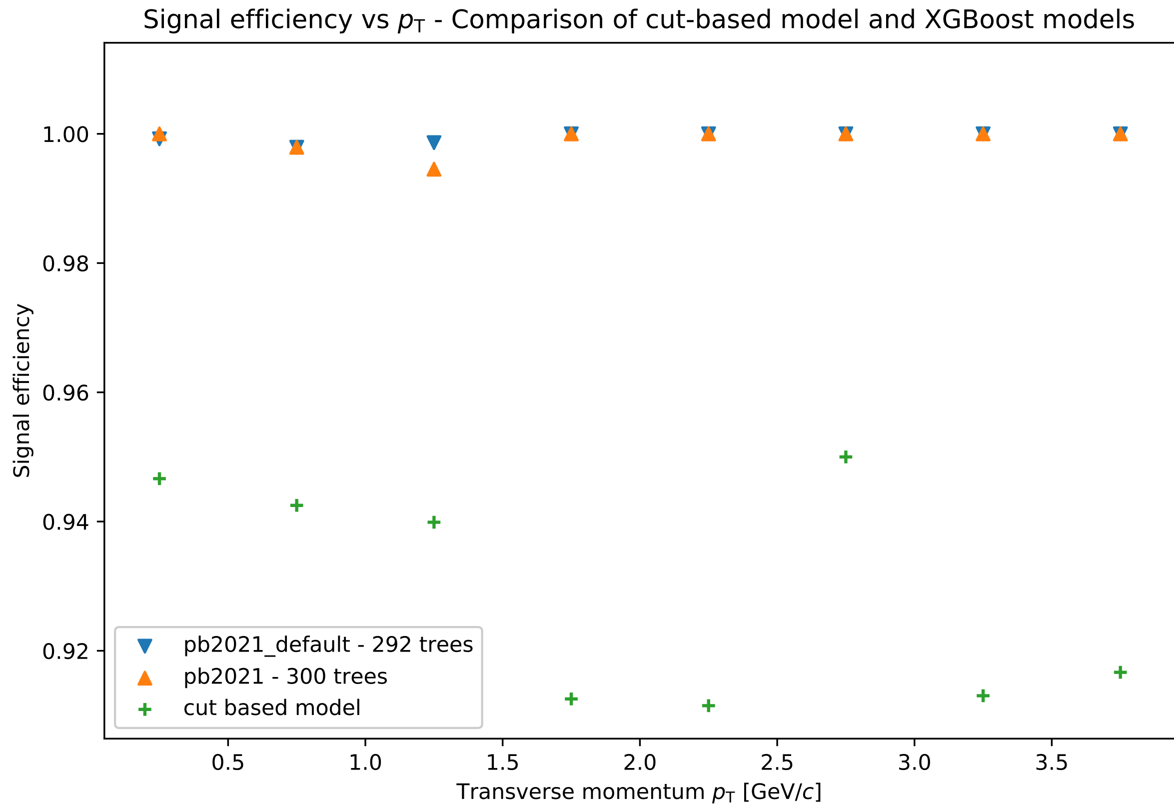


Figure A.5: The signal efficiency of the models evaluated in each transverse momentum bin after the equalized purity. (lead-lead collision system)

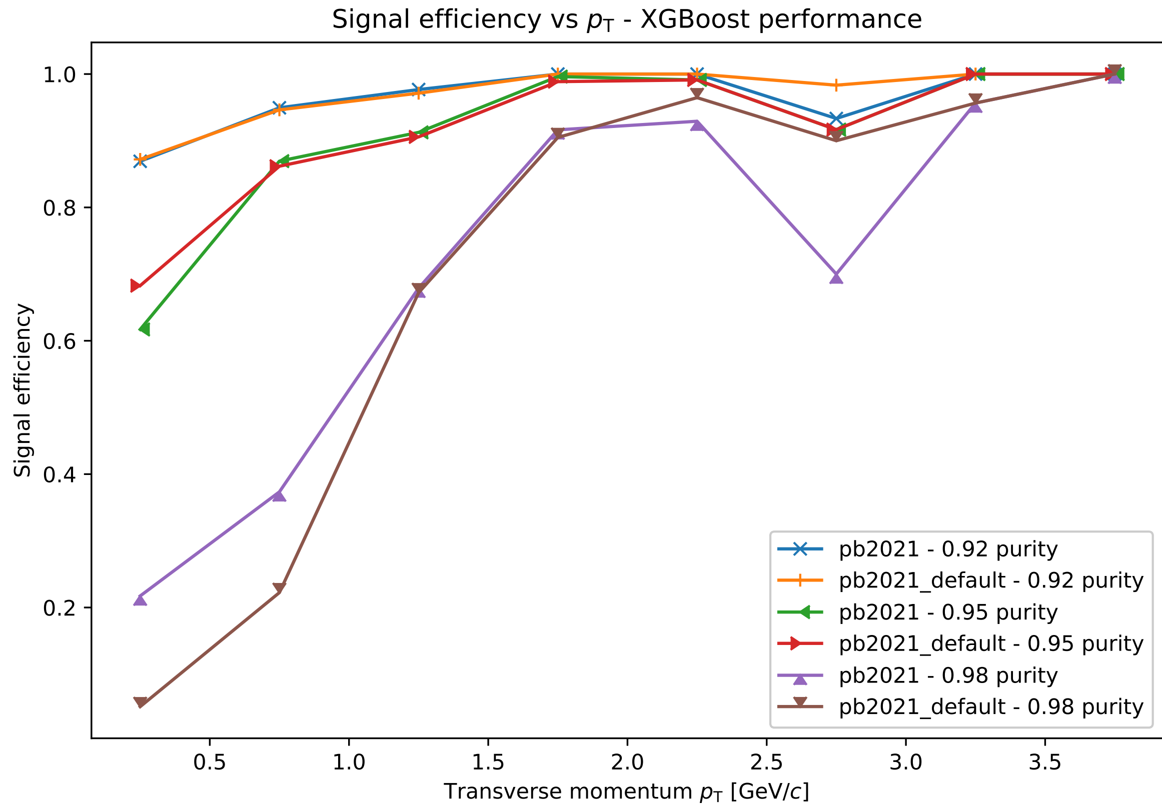


Figure A.6: Signal efficiency of the optimized *pb2021* and the default *pb2021_default* model at 92, 95 and 98% purity. (lead-lead collision system)

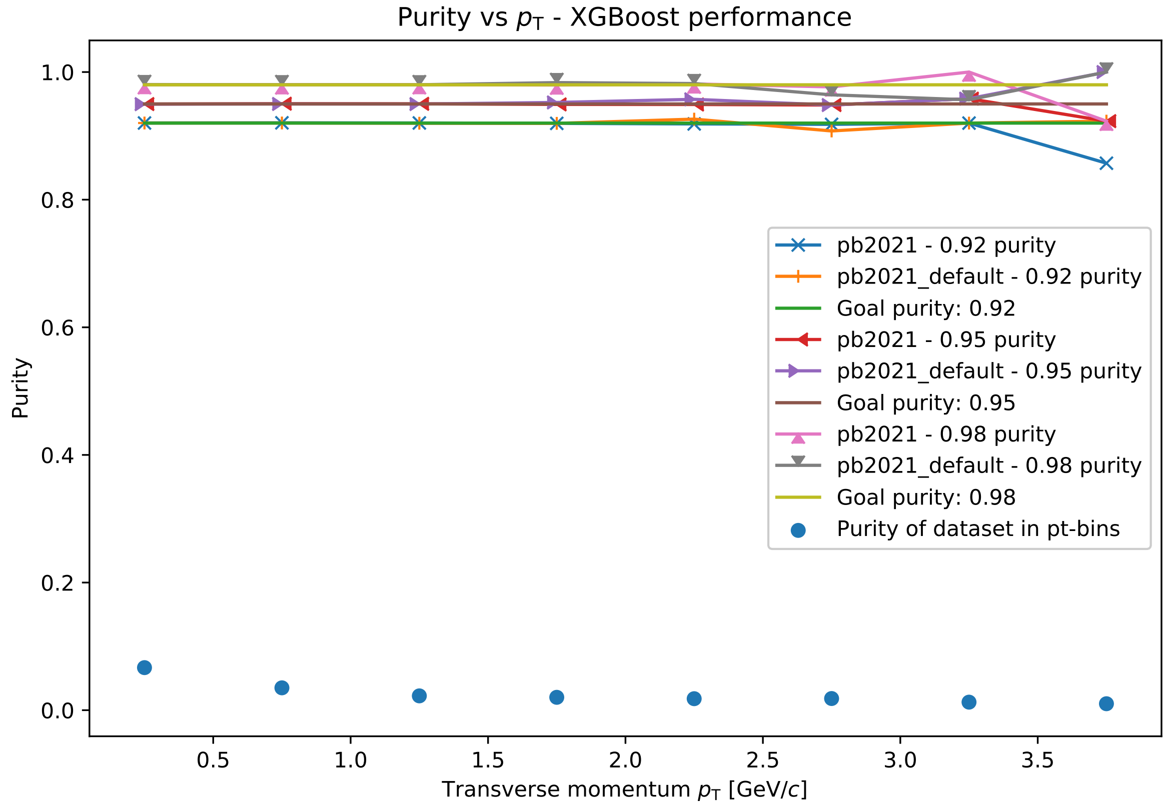


Figure A.7: Adjusting the purity of the XGBoost models to the goal purities in each transverse momentum bin. (lead-lead collision system)

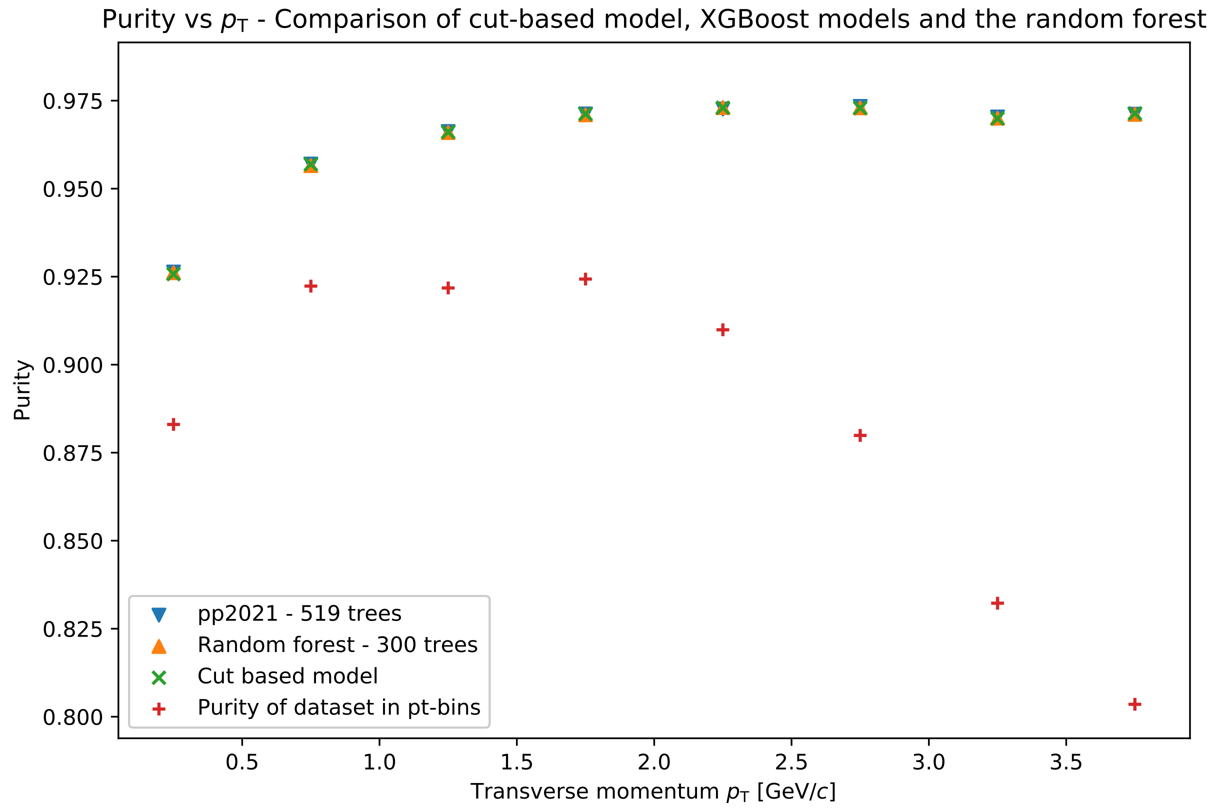


Figure A.8: Adjusting the purity of the models to the purity of the cut-based model in each transverse momentum bin. (proton-proton collision system)

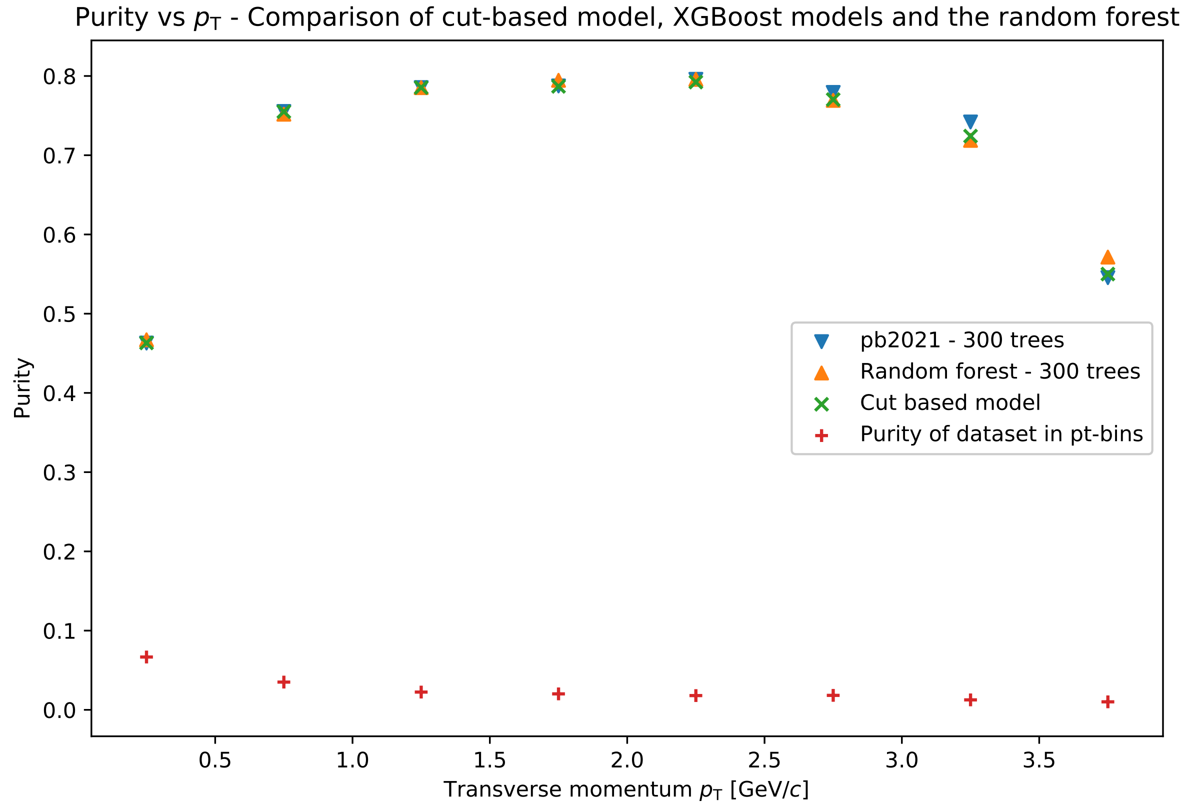


Figure A.9: Adjusting the purity of the models to the purity of the cut-based model in each transverse momentum bin. (lead-lead collision system)

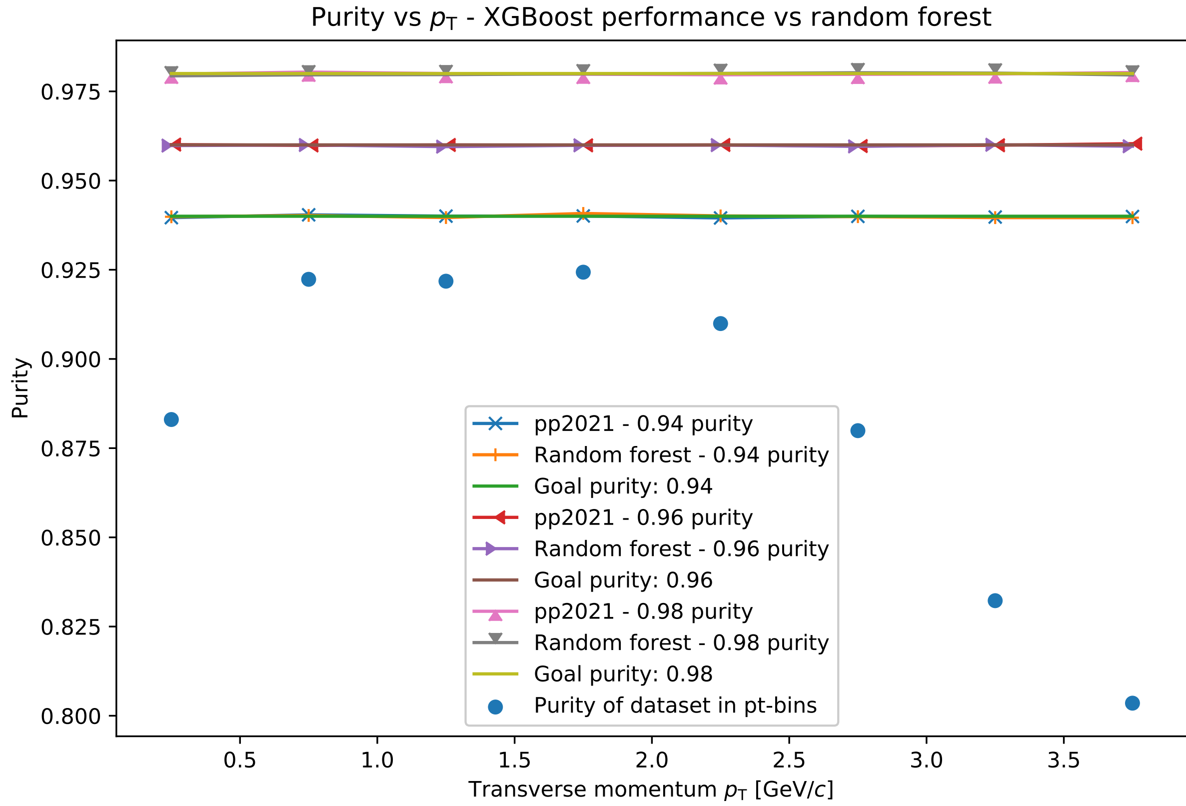


Figure A.10: Adjusting the purity of the models to the purity of the cut-based model in each transverse momentum bin. (proton-proton collision system)

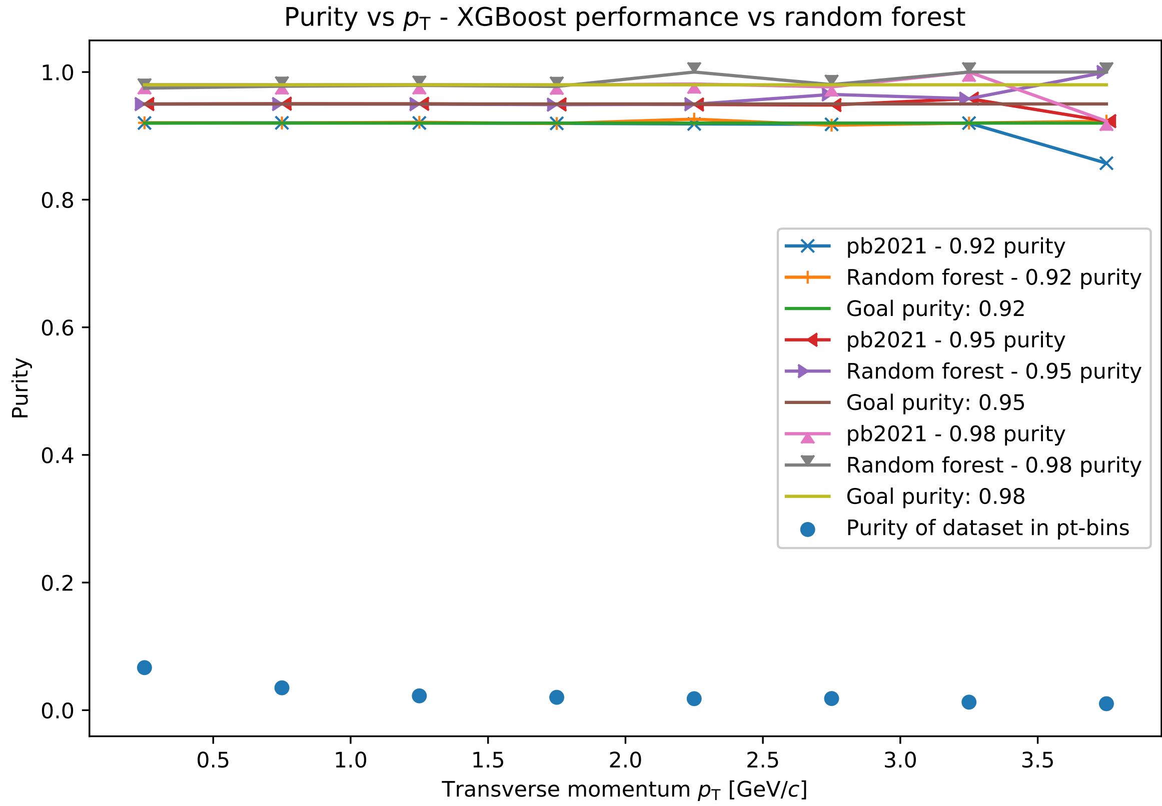


Figure A.11: Adjusting the purity of the models to the purity of the cut-based model in each transverse momentum bin. (lead-lead collision system)

Feature histograms

Proton-proton dataset

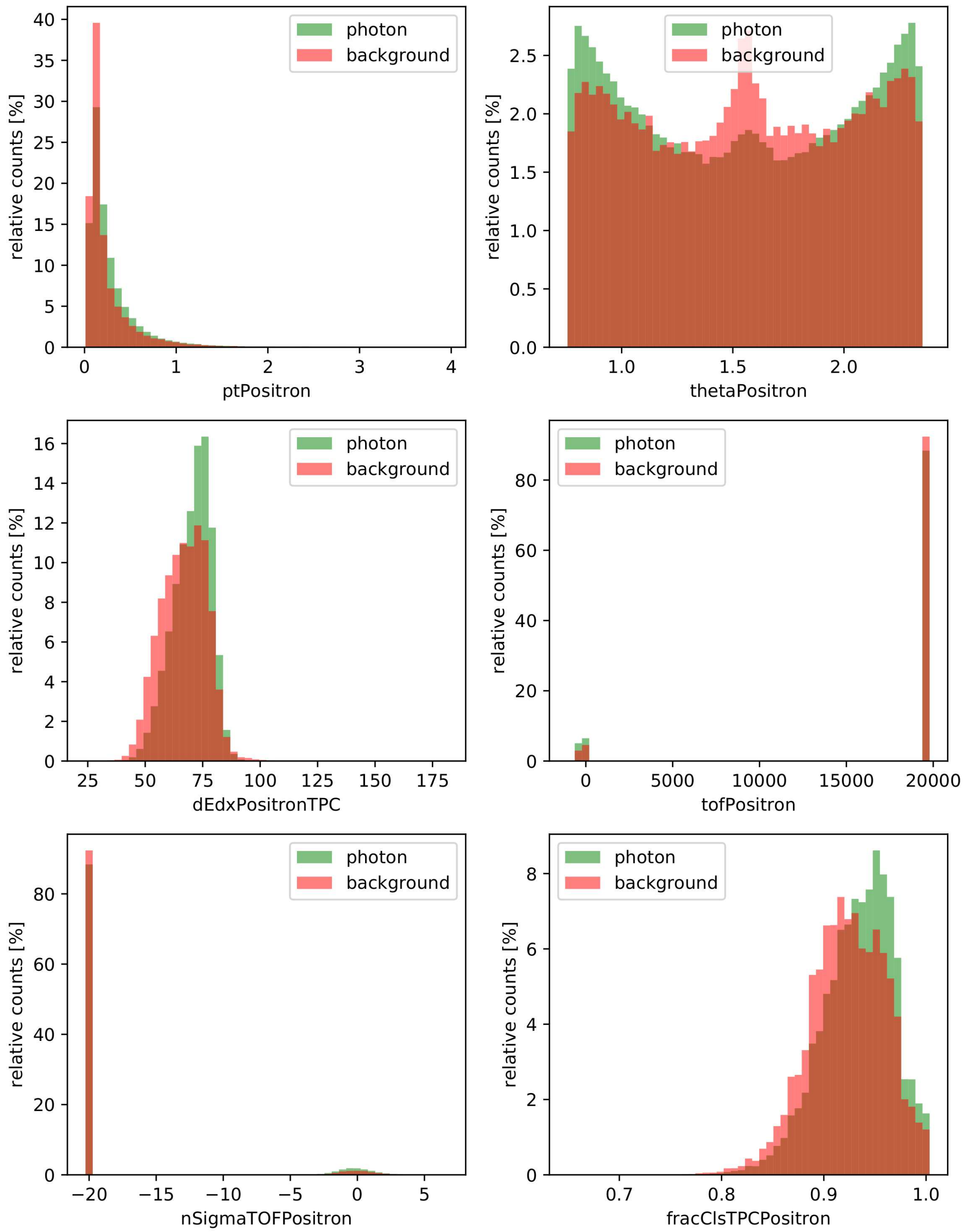


Figure A.12: Feature histograms of the proton-proton dataset.

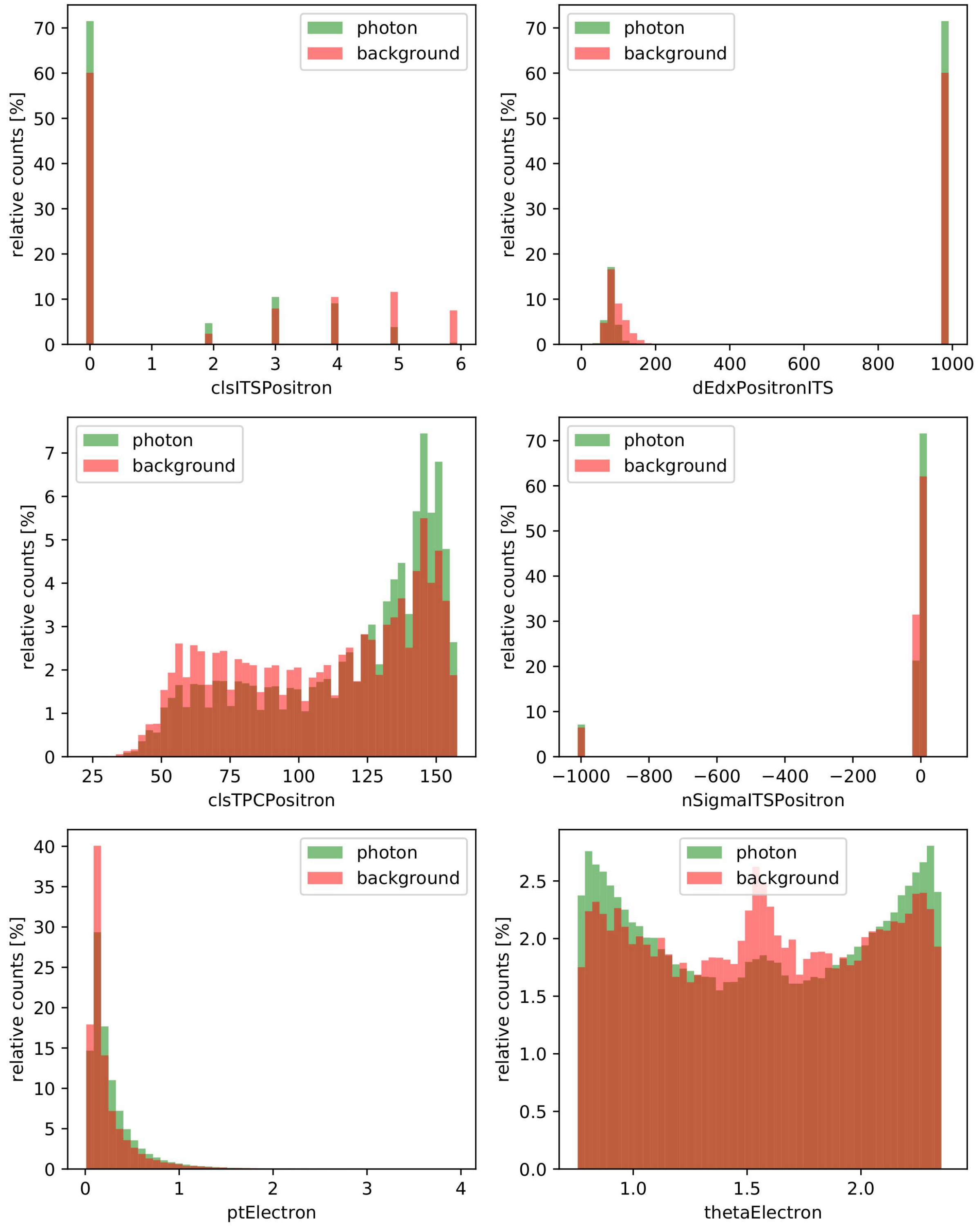


Figure A.13: Feature histograms of the proton-proton dataset.

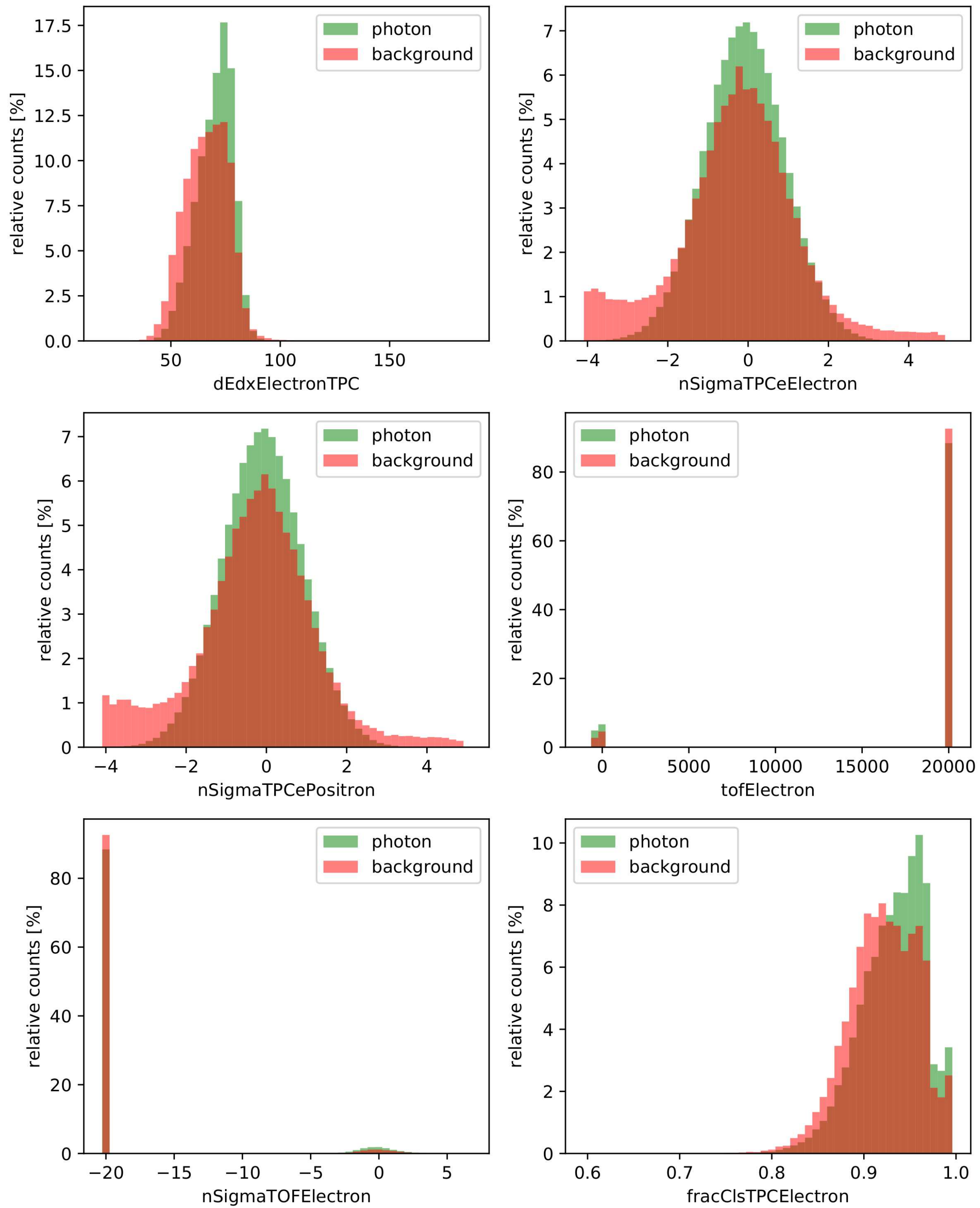


Figure A.14: Feature histograms of the proton-proton dataset.

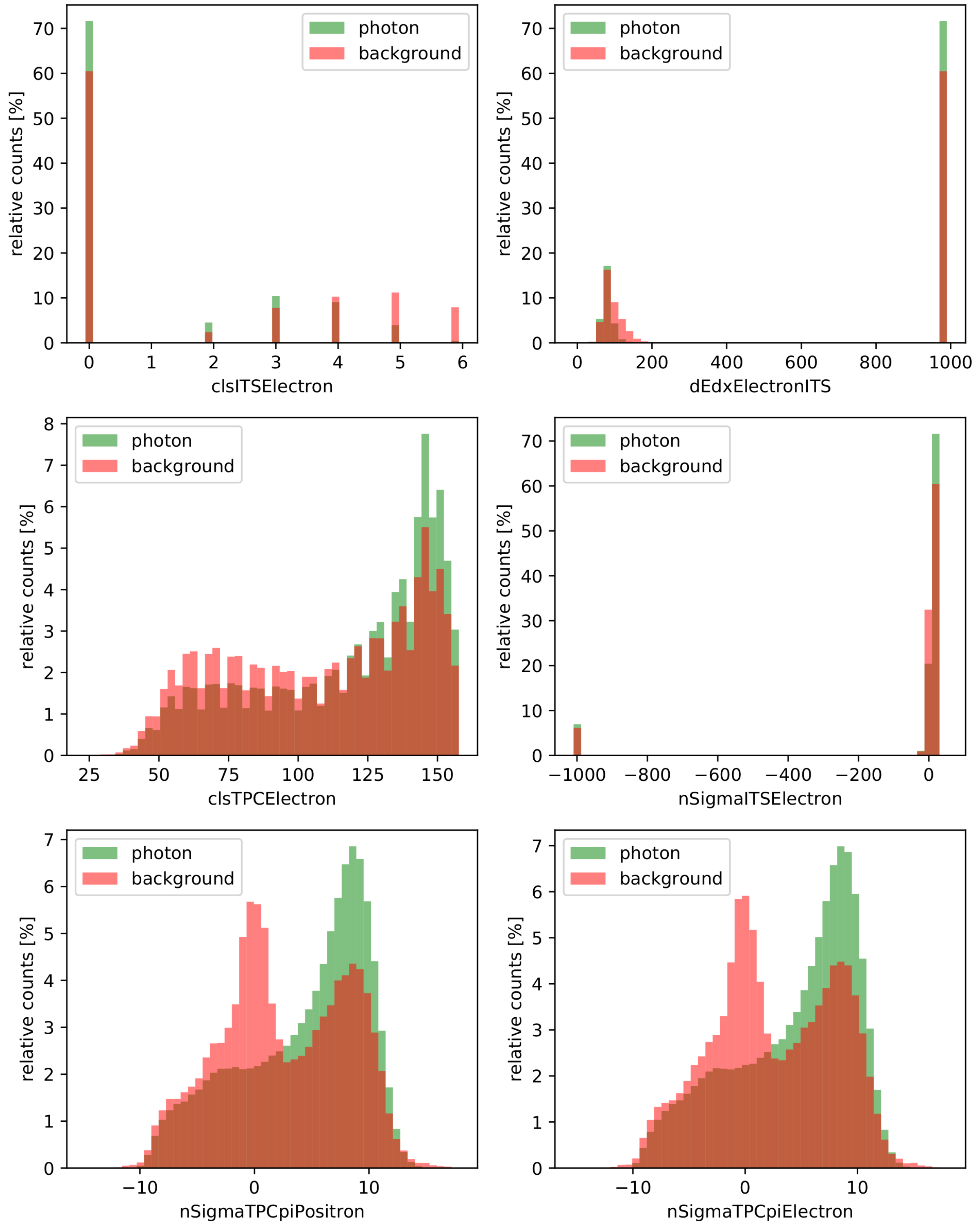


Figure A.15: Feature histograms of the proton-proton dataset.

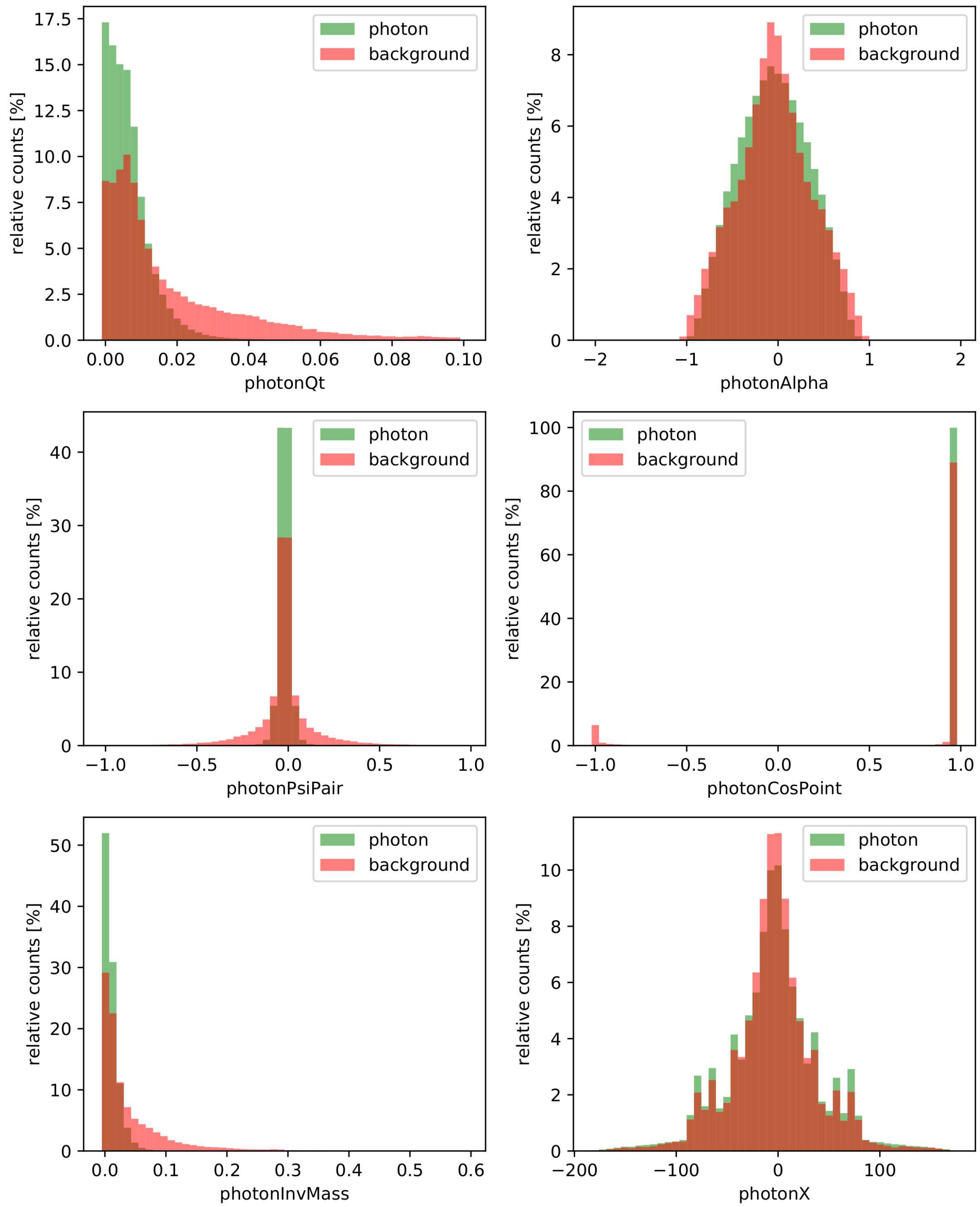


Figure A.16: Feature histograms of the proton-proton dataset.

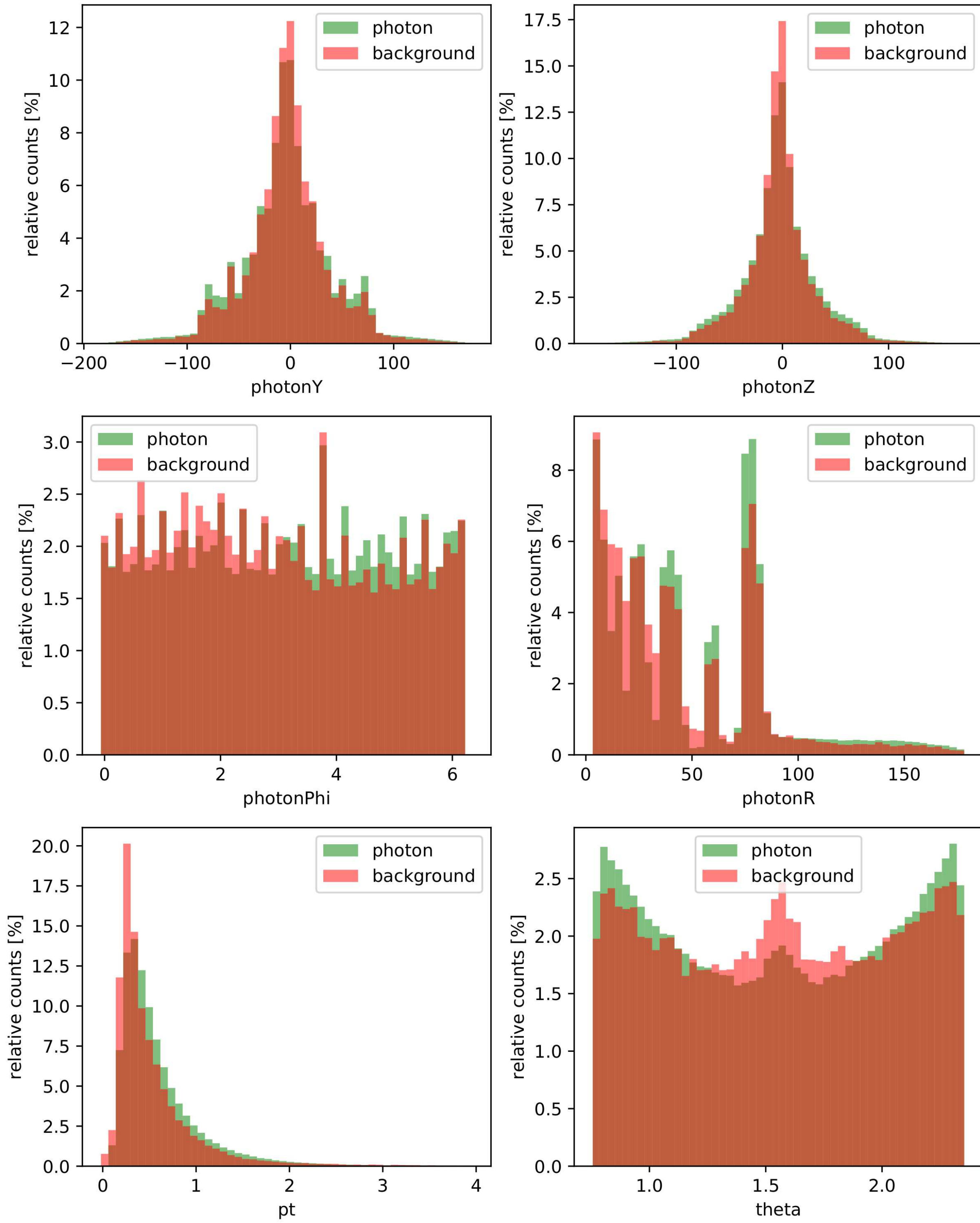


Figure A.17: Feature histograms of the proton-proton dataset.

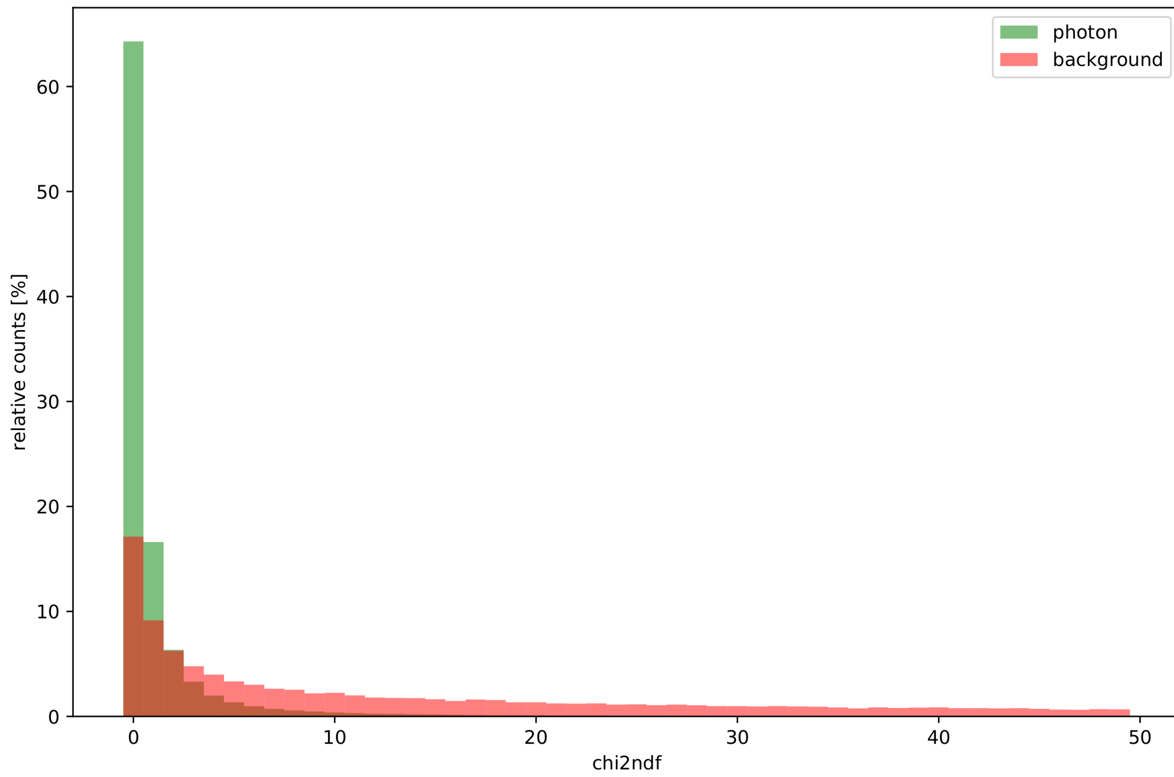


Figure A.18: Feature histograms of the proton-proton dataset.

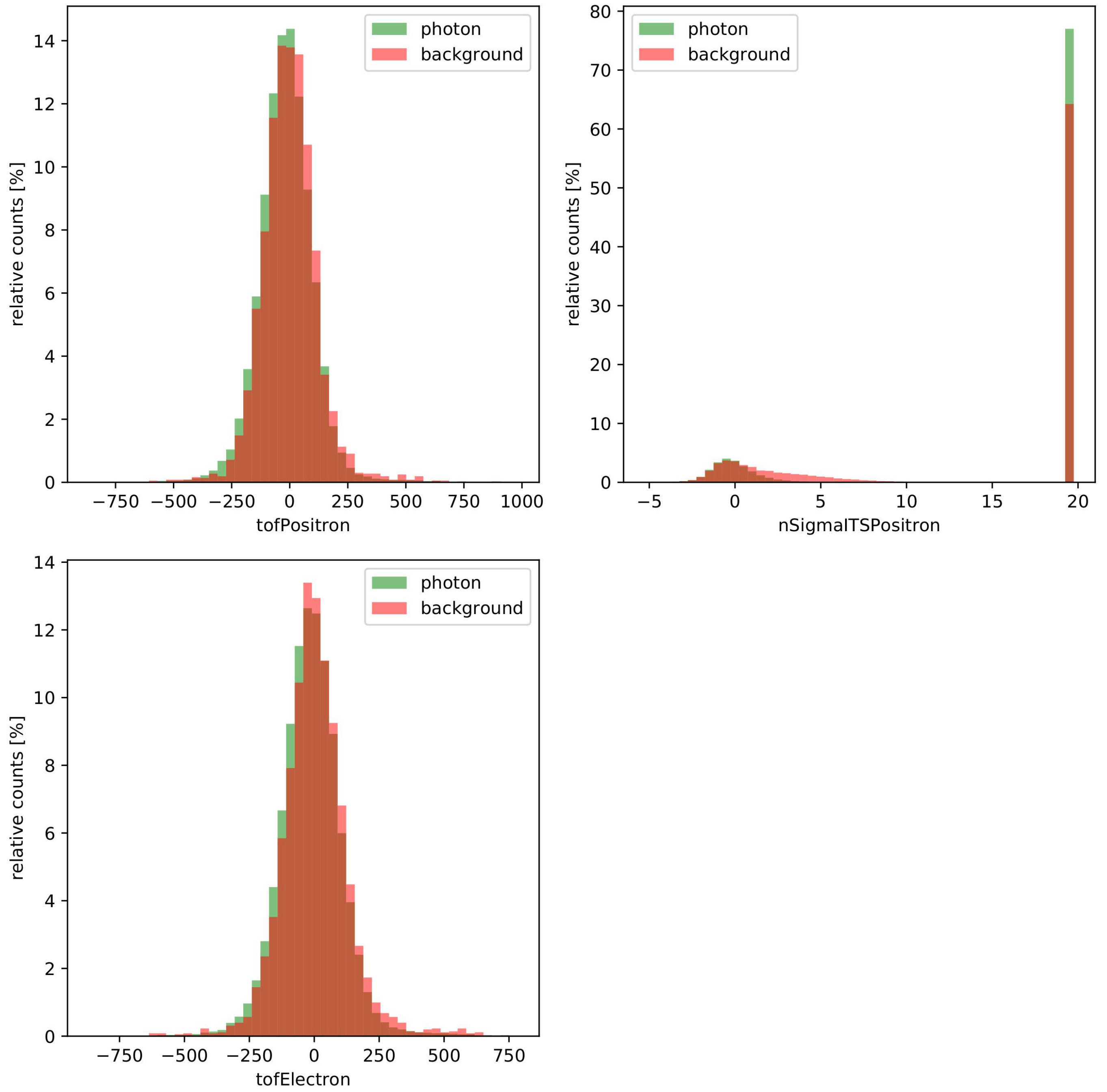


Figure A.19: Selected feature histograms of the proton-proton dataset restricted to particular interval.

Lead-lead dataset

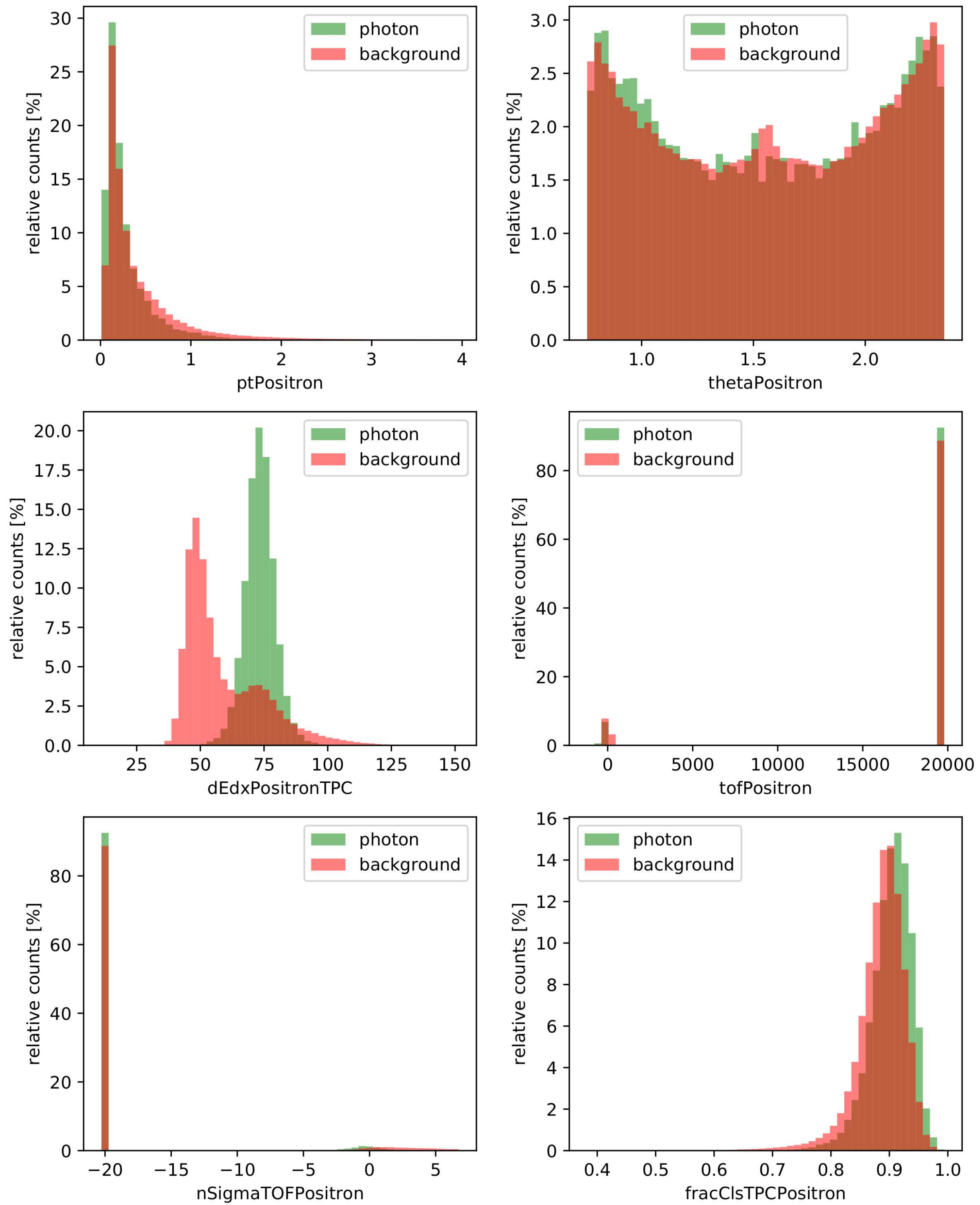


Figure A.20: Feature histograms of the lead-lead dataset.

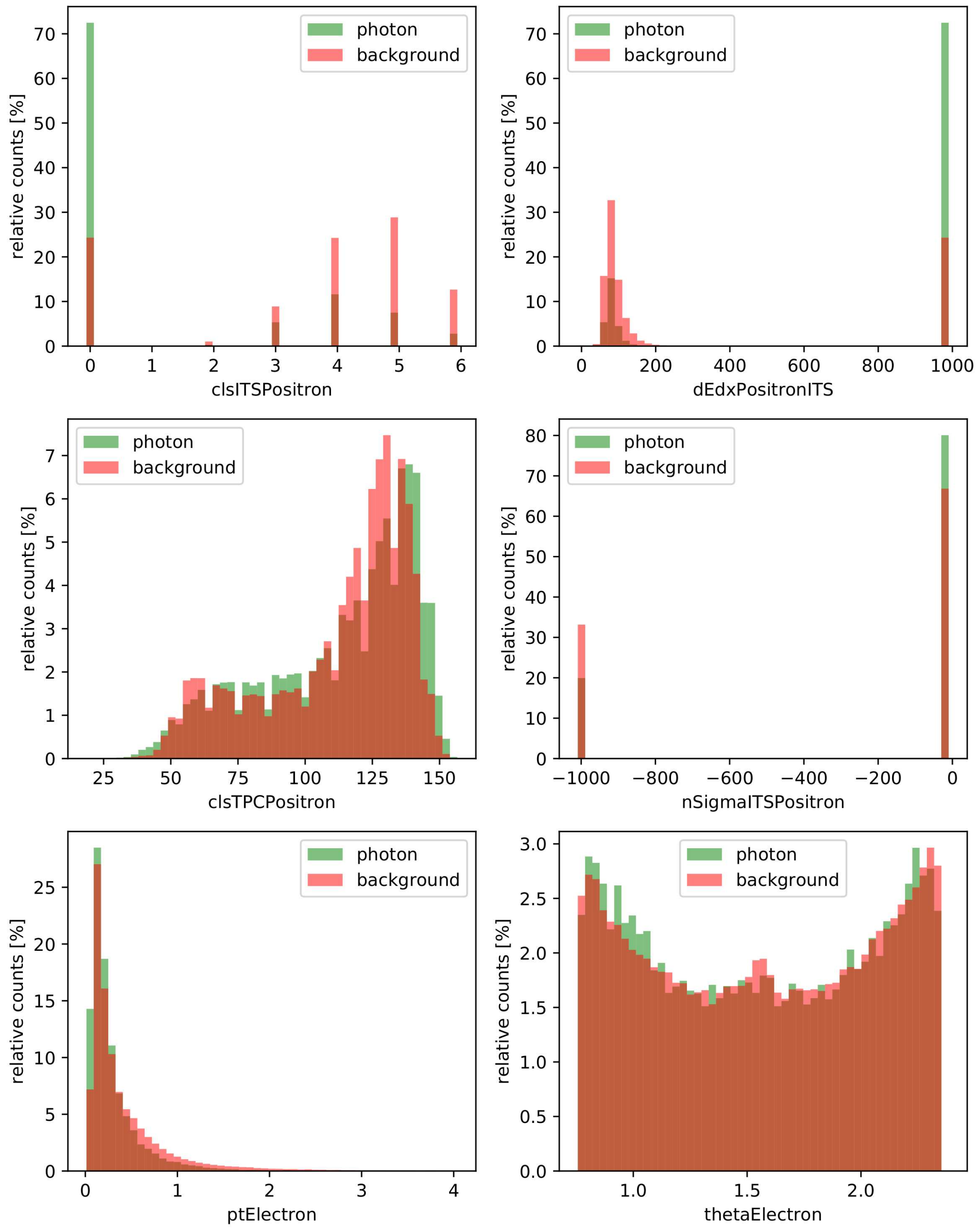


Figure A.21: Feature histograms of the lead-lead dataset.

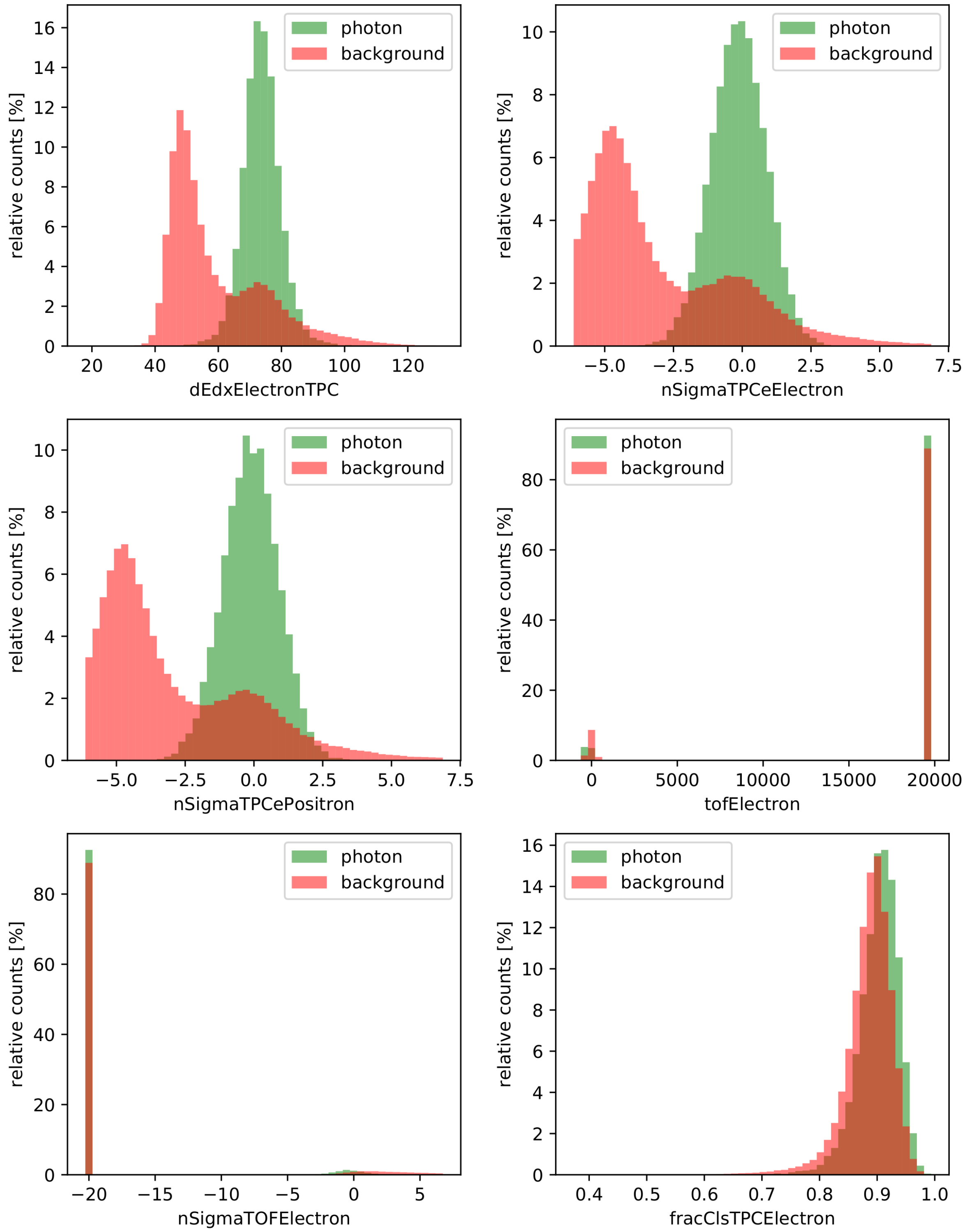


Figure A.22: Feature histograms of the lead-lead dataset.

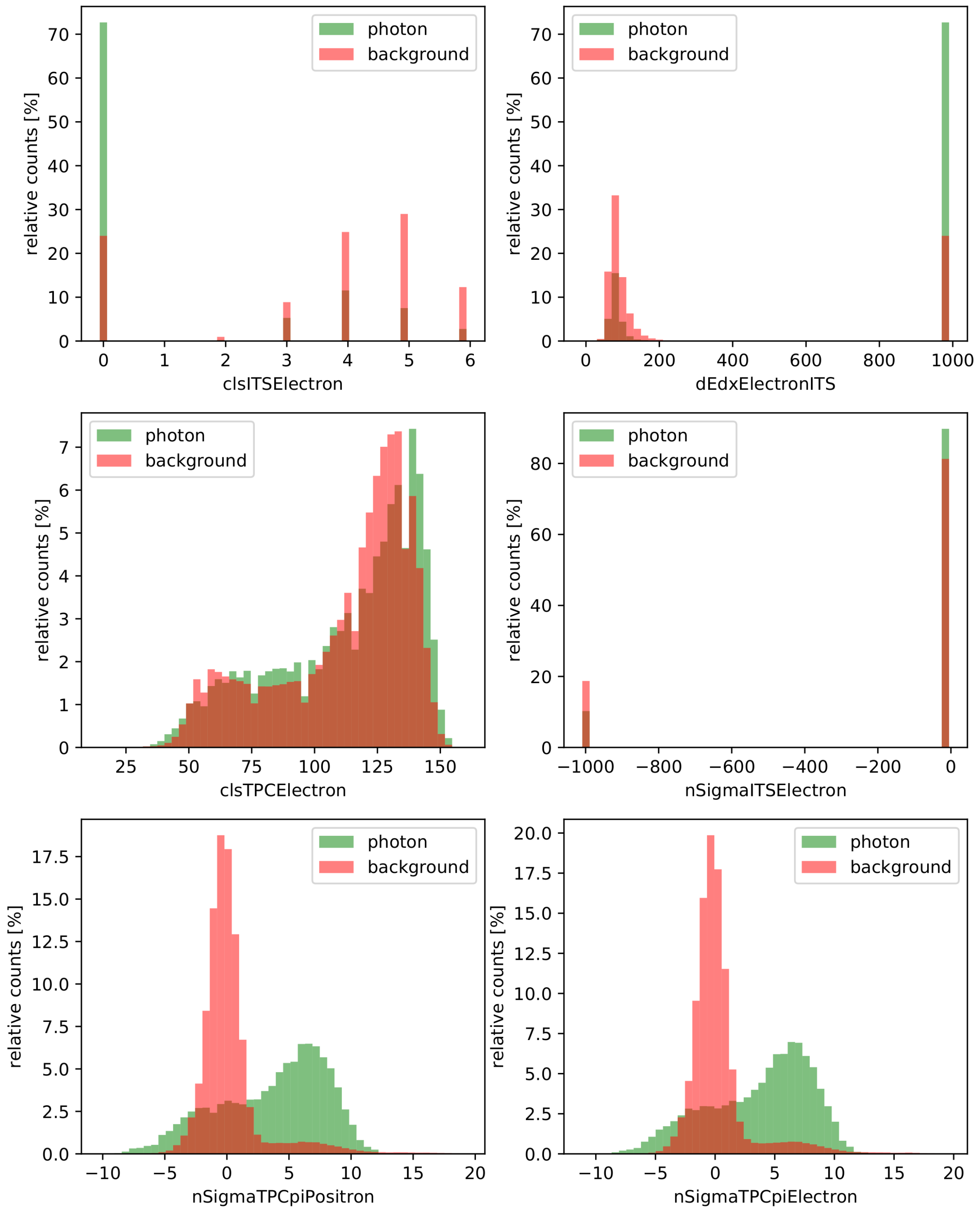


Figure A.23: Feature histograms of the lead-lead dataset.

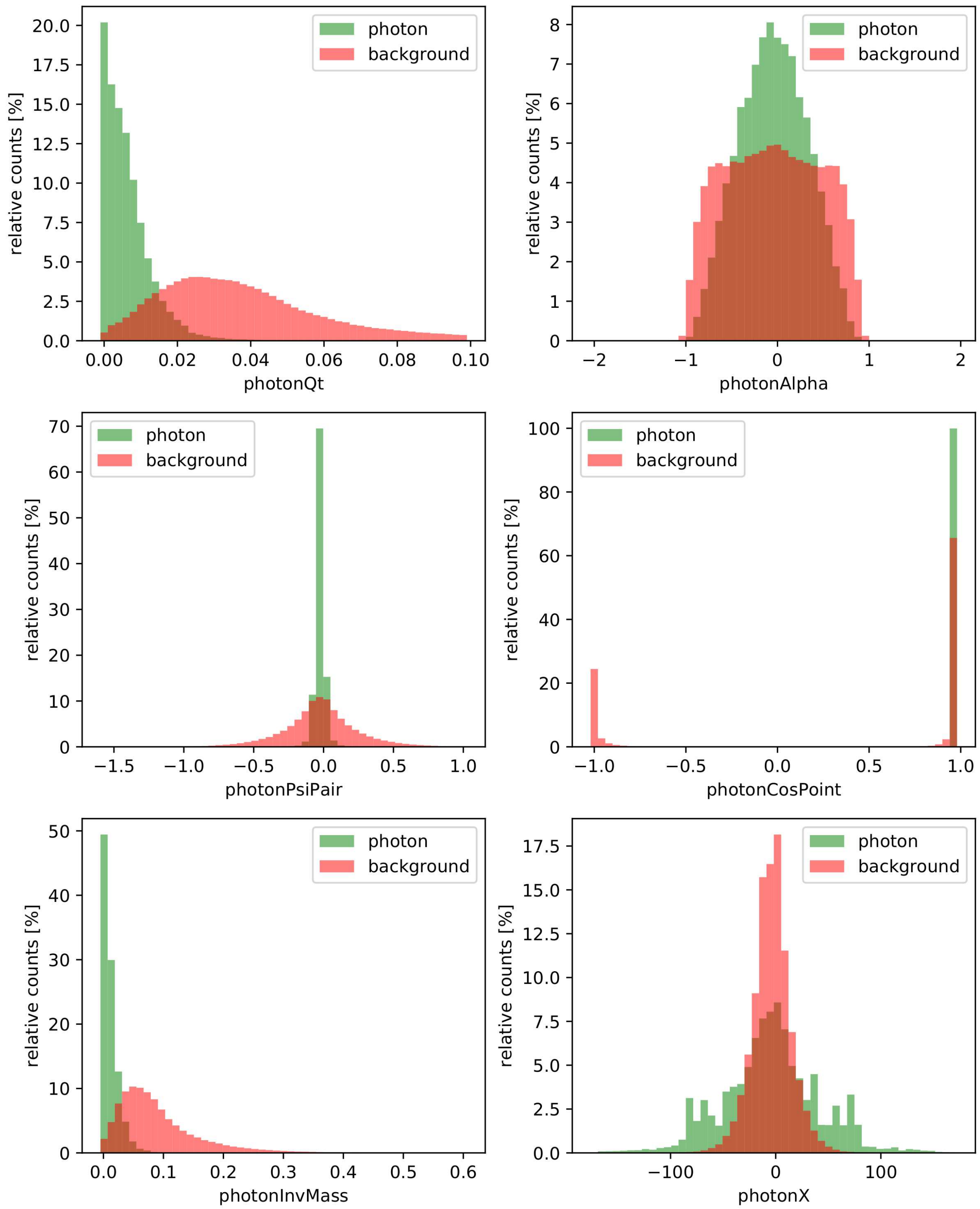


Figure A.24: Feature histograms of the lead-lead dataset.

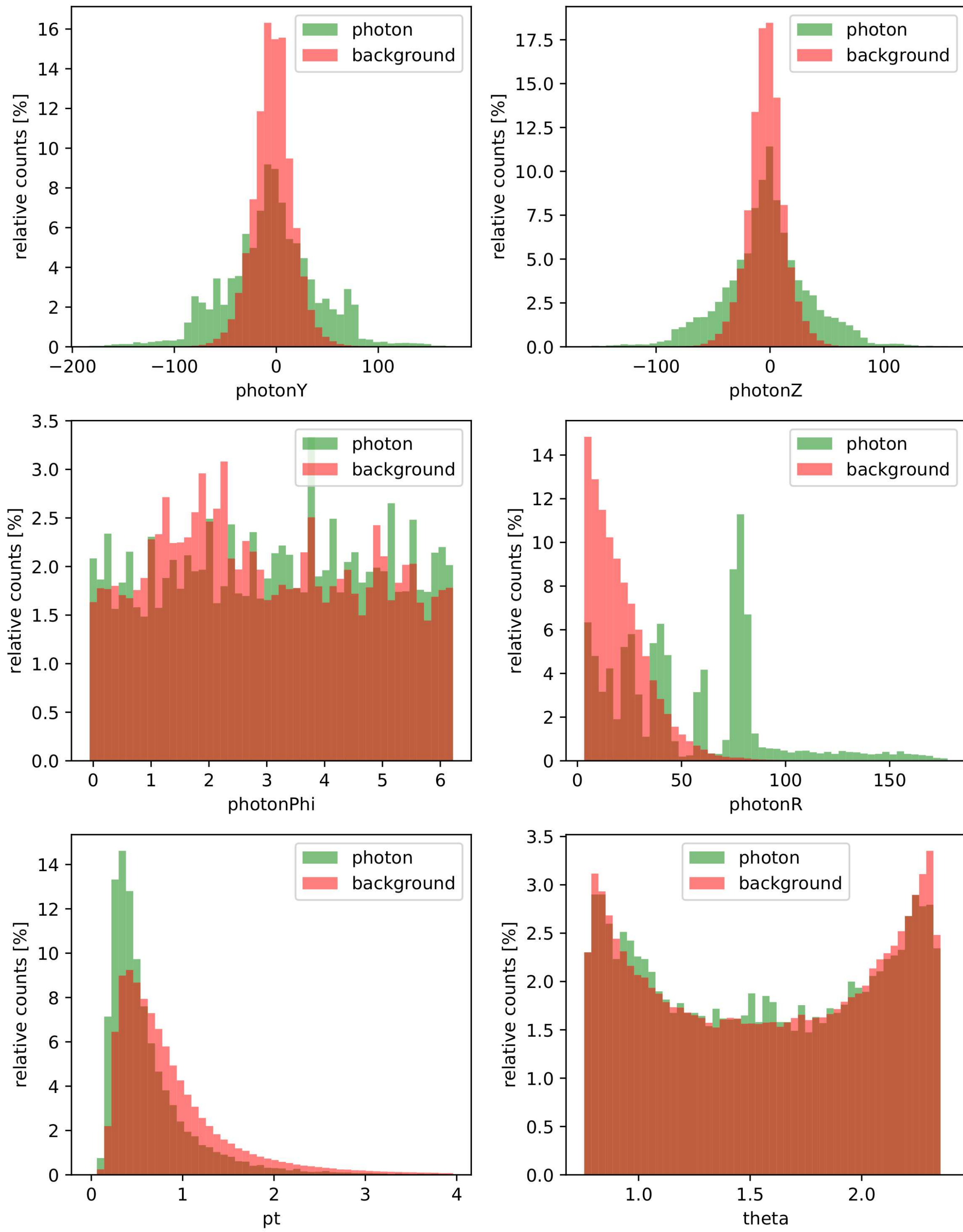


Figure A.25: Feature histograms of the lead-lead dataset.

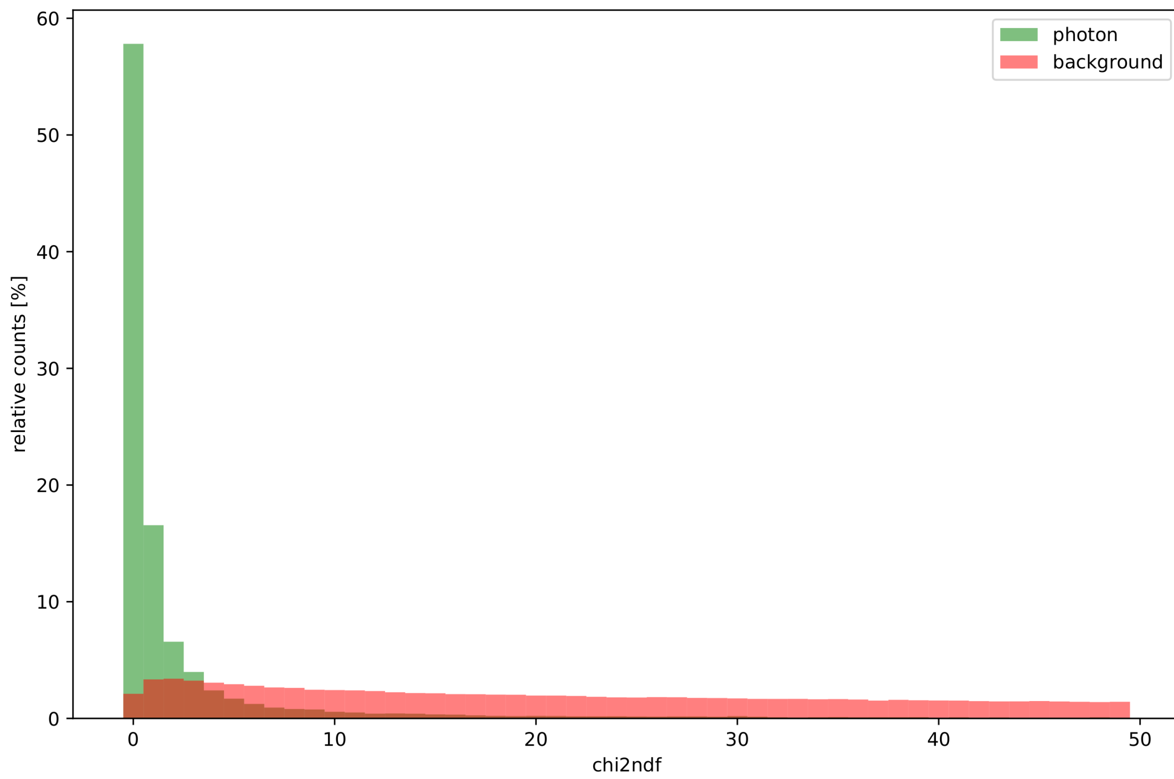


Figure A.26: Feature histograms of the lead-lead dataset.

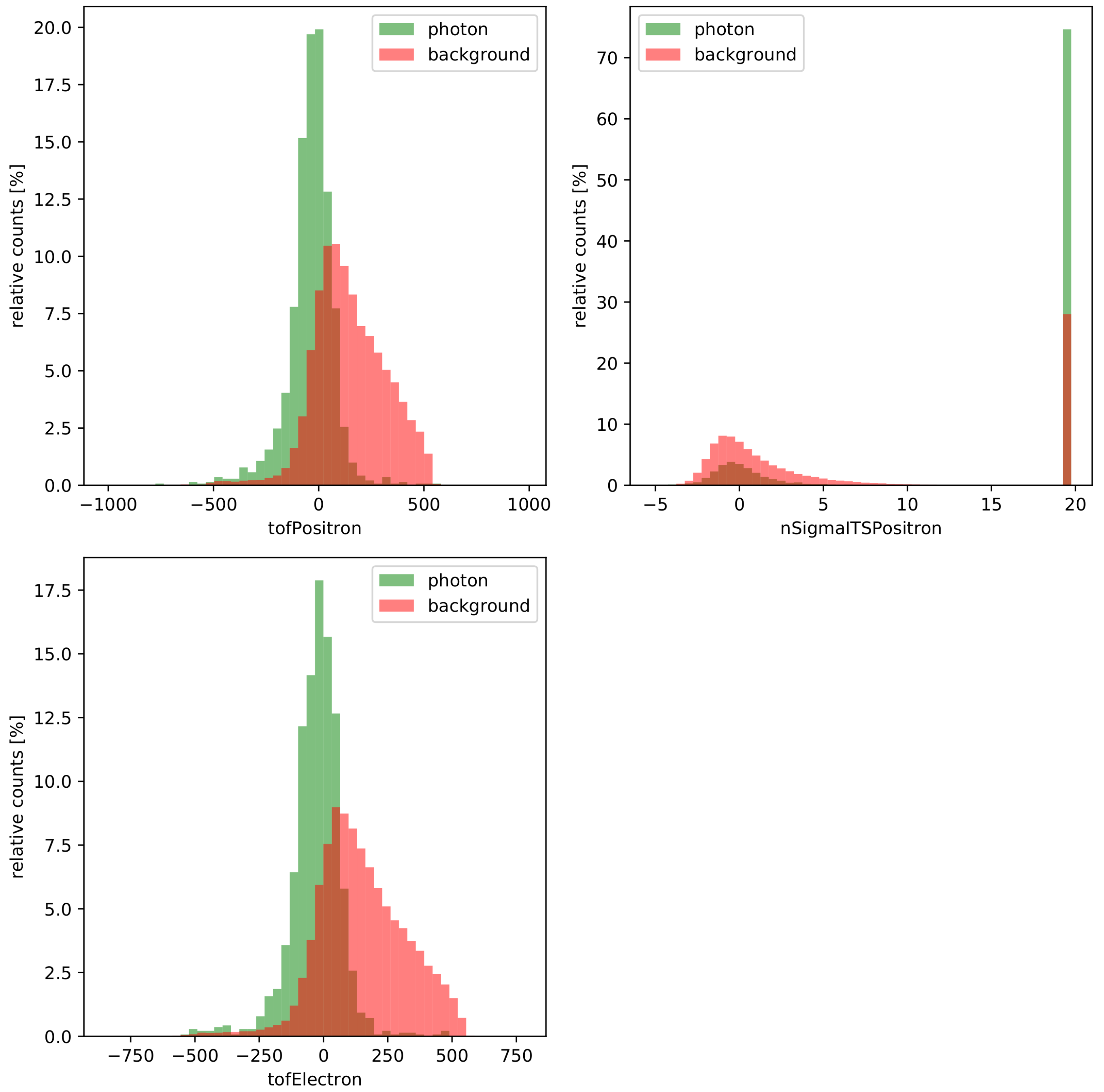


Figure A.27: Selected feature histograms of the lead-lead dataset restricted to particular interval.

Bibliography

[1] D. Boyanovsky, “Phase transitions in the early and the present Universe: from the big bang to heavy ion collisions,” *Phase Transitions in the Early Universe: Theory and Observations*, pp. 3–44, 2 2001. [Online]. Available: http://arxiv.org/abs/hep-ph/0102120

[2] L. Leardini, “Measurement of neutral mesons and direct photons in Pb-Pb collisions at $\sqrt{s_{NN}} = 2.76$ tev with the ALICE experiment at the LHC,” Ph.D. dissertation, 2017.

[3] E. Annala, T. Gorda, A. Kurkela, J. Nättilä, and A. Vuorinen, “Evidence for quark-matter cores in massive neutron stars,” *Nature Physics*, vol. 16, 9 2020.

[4] A. Dainese, “Charm production and in-medium QCD energy loss in nucleus nucleus collisions with ALICE: A Performance study,” 11 2003.

[5] The ALICE Collaboration, “The ALICE experiment at the CERN LHC,” *Journal of Instrumentation*, vol. 3, 8 2008.

[6] B. Abelev, “Upgrade of the ALICE Experiment: Letter Of Intent,” *Journal of Physics G: Nuclear and Particle Physics*, vol. 41, 8 2014.

[7] The ALICE Collaboration, “The ALICE Transition Radiation Detector: Construction, operation, and performance,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 881, 2 2018.

[8] C. Cavicchioli, “Development and commissioning of the pixel trigger system for the alice experiment at the cern large hadron collider,” Ph.D. dissertation, 01 2010.

[9] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009. [Online]. Available: http://www-stat.stanford.edu/~tibs/ElemStatLearn/

[10] J. H. Friedman, “Greedy function approximation: A gradient boosting machine.” *The Annals of Statistics*, vol. 29, 10 2001.

[11] Distributed Machine Learning Community, “xgboost,” 2020. [Online]. Available: https://github.com/dmlc/xgboost/blob/master/NEWS.md

[12] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System.” ACM, 8 2016.

[13] A. Alves, “Stacking machine learning classifiers to identify Higgs bosons at the LHC,” *Journal of Instrumentation*, vol. 12, 5 2017.

[14] D. Nielsen, “Tree Boosting With XGBoost Why Does XGBoost Win "Every" Machine Learning Competition?” 2016. [Online]. Available: http://pzs.dstu.dp.ua/DataMining/boosting/bibl/Didrik.pdf

[15] P. Zyla *et al.*, “Review of Particle Physics,” *PTEP*, vol. 2020, no. 8, p. 083C01, 2020.

[16] A. P. Kalweit, “Energy Loss Calibration of the ALICE Time Projection Chamber,” 2008. [Online]. Available: https://files.transtutors.com/cdn/uploadassignments/1499962_2_article-1.pdf

[17] E. Kryshen, D. Ivanischev, D. Kotov, M. Malaev, V. Riabov, and Y. Riabov, “Thermal photon and neutral meson measurements using photon conversion method in the MPD experiment at the NICA collider,” 2020. [Online]. Available: https://indico.jinr.ru/event/1469/contributions/9865/attachments/8180/12209/2020-10-22-kryshen-photons.pdf

[18] M. C. Williams, “Particle identification using time of flight,” *Journal of Physics G: Nuclear and Particle Physics*, vol. 39, 2012.

[19] The ALICE Collaboration, “Measurement of the low-energy antideuteron inelastic cross section,” *Phys. Rev. Lett.*, vol. 125, p. 162001, Oct 2020. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.125.162001

[20] T. K. Ho, “Random decision forests.” IEEE Comput. Soc. Press.

[21] Prof. Dr. Klaus Reygers, Private Communication, 2021.

Acknowledgment

After five years of graduate studies at the University of Heidelberg, I am finally at a point in life, which seemed so far away from me. I encountered many challenges, which taught me that I am still an entrepreneur on the path of life. But I was not alone on this path for which I am honestly grateful. People come and go, but every encounter contributed a part, which became the best of me. I would express my sincere gratitude for apl. Prof. Dr. Klaus Reygers. It is an invaluable experience to write my thesis in the ALICE group. For all the time and guidance, I am thankful for the support of Martin Kroesen. Also, I would like to thank Prof. Dr. Silvia Masciocchi for being the second referee for this thesis. Lukas Berger, thank you for all your support and time to proofread my thesis.

Declaration

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgment, the work presented is entirely my own.

April 27, 2021, Heidelberg

Xuan-Xuyen Nguyen