Specially designed random forest loss function for high energy physics

with data from the CERN's ATLAS experiment

Dovydas Sprindys

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 1, 2022

Abstract

The purpose of the ATLAS experiment at CERN is to provide a better understand of the underlying principles of fundamental particles and to potentially discover new ones, such as dark matter. The process of doing so is long and difficult, requiring different types of expertise. One part of the process is to investigate the data recorded and determine whether deviations from known physics can be observed.

In this thesis, different loss functions, including ones that are custom designed, will be applied to machine learning algorithms to assess whether they can improve the separation of data that has a potential to contain information about new particles versus the data that contains only known physics. A paper presenting the findings of this thesis is in a preparation with the intention of being published.

Acknowledgements

Working on a project related to CERN and ATLAS has been as fascinating and intriguing as it has been instructive. I would like to thank my supervisors - Trygve Buanes and Therese Berge Sjursen for providing this opportunity, in addition to valuable insight, guidance and feedback throughout the entire project. In addition, special thanks to Trygve Buanes for providing ideas and help with the development of the theory behind the custom loss function.

Table of contents

Glossary **6**

1 Introduction **11**

2 Background **12**
2.1 Physics Behind ... 12
2.1.1 Standard model ... 12
2.1.2 Jets ... 14
2.1.3 Taus ... 16
2.2 Machine Learning and CERN ... 16

3 Thesis Outline **17**
3.1 Research questions ... 17
3.2 Research Methods ... 17
3.3 Plan For Evaluation ... 18
3.4 Thesis structure ... 18
3.5 Expectations ... 18
3.6 Relevant Research ... 19

4 The ATLAS Experiment **20**
4.1 LHC ... 20
4.2 Particle detectors ... 21
4.3 The ATLAS detector ... 22
4.3.1 The Inner Detector ... 22
4.3.2 Calorimeter ... 23
4.3.3 Muon Spectrometer ... 24
4.3.4 Magnet System ... 24
4.3.5 Cross-Section ... 25
4.3.6 Trigger Systems ... 26

5 Data **27**
5.1 Simulated Data ... 27
5.2 Real Data ... 28
5.3 Data files ... 28
5.4 Features ... 29
5.5 Data cuts ... 30
5.6 Feature and simulated data validation ... 31

6 Machine Learning **35**
6.1 Decision Trees ... 36
6.1.1 Variance reduction ... 36
6.1.2 Custom loss function ... 37
6.2 Random forest ... 39
6.3 Implementation ... 41

6.4 Implementation and performance assessment 46
6.5 Development Tools 54

7 Decision Tree Analysis **56**
7.1 Hyperparameter testing and model optimization 56
7.1.1 Process 56
7.1.2 Results 57
7.2 Observations 58

8 Random Forest Analysis **61**
8.1 Random Forest Using Classification 61
8.1.1 Theory 61
8.1.2 Process 61
8.1.3 Results 62
8.2 RF performance fluctuation testing 65
8.2.1 The need for consistency 65
8.2.2 Intermediate results 66
8.2.3 Potential Solutions 67
8.2.4 Preliminary dimensionality reduction 68
8.2.5 Increasing the number of estimators to 30 70
8.2.6 Balanced dataset and alternative weighting 70
8.2.7 Feature importance 72
8.2.8 Increasing the number of estimators to 60 74
8.2.9 Remarks 76
8.3 Hyperparameter testing using reduced fluctuations 78
8.3.1 Results 78
8.4 Increasing the number of features 83
8.4.1 Dimensionality reduction utilizing feature importance 85
8.5 Feature importance validation 87
8.6 Alternative custom loss function 92
8.7 Different signal files 95
8.8 Performance using data from different data collection periods 98
8.8.1 Data Collection Period $\boldsymbol{d}$ 98
8.8.2 Data Collection Period $\boldsymbol{e}$ 99
8.8.3 Remarks 100
8.9 Errors 101
8.10 Unbalanced datasets 102

9 Conclusions **104**

10 Further Work **105**

A Decision tree hyperparameter testing

B Feature validation plots

C Custom loss split verification

Glossary

ML Machine Learning

CERN The European Organization for Nuclear Research

LHC Large Hadron Collider

ATLAS Particle detector experiment at the LHC

HEP High Energy Physics

NN Neural Networks

RF Random Forest

List of Figures

1	The Standard Model [4]	12
2	Electron and positron producing a quark, anti-quark pair [6]	14
3	Electron and positron producing a quark and an antiquark in addition to a gluon [6]	14
4	A typical three jet event visualized [6]	15
5	A typical two jet event visualized[6]	15
6	Hadronization and jet formation [6]	15
7	Tau Decay	16
8	The Aerial View of the Underground LHC [10]	20
9	CERN's Accelerator Complex [11]	21
10	The ATLAS Detector [12]	22
11	ATLAS Inner Detector [13]	22
12	ATLAS Calorimeter [15]	23
13	ATLAS Muon Spectrometer [17]	24
14	CERN's Accelerator Complex [19]	24
15	Event Cross-Section in a computer generated image of the ATLAS detector [20].	25
16	Feature validation for feature *met*	32
17	Feature validation for feature *METoverPtMean*	32
18	Feature validation for feature *METSig*	33
19	Feature validation for feature *jet_1_pt*	33
20	Feature validation for feature *sumMT*	34
21	Machine learning vs Traditional programming [24]	35
22	Decision tree node split visualized	38
23	Sensitivity plot of decision tree model, implemented using a custom implementation	47
24	Sensitivity plot of decision tree model, implemented using *sklearn* library	47
25	Decision tree graph of decision tree model, implemented using custom implementation	48
26	Decision tree graph of decision tree model, implemented using *sklearn* library	49
27	Comparison of top 2 features with the highest sensitivity increase achieved	51
28	Threshold reduction performance, $n = 10$	52
29	Threshold reduction performance, $n = 50$	52
30	Threshold reduction performance, $n = 100$	52
31	Threshold reduction performance, $n = 300$	52
32	Close-up of loss function values calculated for thresholds	53
33	Loss function value plot for feature *jet_1_mtMet*, $n = 100$	54
34	Sensitivity plots using different loss functions	57
35	Example of a decision tree graph using custom *sum* loss function	60

36 Sensitivity comparison of RF models using *custom sum* and variance reduction loss functions for different minimum splits, using depth 3 ... 63
37 Sensitivity comparison of RF models using *custom sum* and variance reduction loss functions for different minimum splits, using depth 6 ... 64
38 Deviations of peak sensitivities for RF models using same parameters ... 66
39 Deviations of peak sensitivities for RF models using 12 most important features ... 69
40 Deviations of peak sensitivities for RF models using 30 estimators 70
41 Deviations of peak sensitivities for RF models using balanced dataset sampling ... 71
42 An example of initial feature importance plots ... 73
43 Deviations of peak sensitivities for RF models using 60 estimators 74
44 Sensitivity deviations using *custom non-sum* loss funcion with 30 estimators ... 75
45 Sensitivity deviations using *custom non-sum* loss funcion with 60 estimators ... 76
46 RF model deviations in peak sensitivities using different loss functions and corresponding best hyperparameter combinations ... 81
47 Feature importance distributions for different loss functions ... 82
48 Deviations of peak sensitivities for RF models with reduced features (-6) ... 86
49 Feature importance distributions for different loss functions using *ttbar* and *wtaunu* data files ... 88
50 Feature importance validation *jet_1_mtMet* ... 90
51 Feature importance validation *jet_2_mtMet* ... 90
52 Feature importance validation *METSig* ... 90
53 Feature importance validation *ht* ... 90
54 Feature importance validation *tau_1_mtMet* ... 90
55 Feature importance validation *tau_1_pt* ... 90
56 Feature distributions of best performing features for *wtaunu* and *ttbar* datasets ... 90
57 Feature importance validation *ele_n* ... 91
58 Feature importance validation *jet_1_n* ... 91
59 Feature importance validation *mu_n* ... 91
60 Feature importance validation *jet_1_width* ... 91
61 Feature importance validation *tau_1_ntracks* ... 91
62 Feature importance validation *tau_n* ... 91
63 Feature distributions of worst performing features for *wtaunu* and *ttbar* datasets ... 91
64 RF classification models' fluctuations in peak sensitivities using alternative custom loss functions, as described in Equation (6)... 93
65 Feature distribution comparison of custom *sum* loss and alternative loss functions ... 94

66 Deviations of peak sensitivities using GG_1700 signal file 95
67 Deviations of peak sensitivities using GG_1700 signal file using unbalanced dataset 97
68 Deviations of peak sensitivities using *d* dataset 99
69 Deviations of peak sensitivities using *e* dataset 100
70 Deviations of peak sensitivities using *a* dataset, with error bars . 102
71 Deviations of peak sensitivities using *a* dataset and increased signal in the training data, with error bars 103

List of Tables

1 Description of datafiles 28
2 List of active features 30
3 Data cuts applied in the ML analysis 31
4 Data cuts applied in feature validation plots 31
5 Peak sensitivities of variance reduction 57
6 Peak sensitivities of custom sum loss function 58
7 Peak sensitivities of custom non-sum loss function 58
8 High-level performance comparison between custom loss function and variance reduction 65
9 Deviations of peak sensitivities for RF models using same parameters 67
10 Deviations of peak sensitivities for RF models using 12 most important features 68
11 Hyperparameter test results using weighted custom *sum* loss function 78
12 Hyperparameter test results using unweighted custom *sum* loss function 78
13 Hyperparameter test results using custom *non-sum* loss function . 79
14 Hyperparameter test results using variance reduction with 30 estimators and 25% train data per tree 79
15 Hyperparameter test results using variance reduction with 60 estimators and 12.5% train data per tree 79
16 RF model performance comparison using different loss functions and corresponding best *depth* and *minimum samples split* hyperparameters 80
17 RF model performance comparison with different amount of features selected per tree, using *custom non-sum* custom loss function. 83
18 RF model performance comparison with different amount of features selected per tree, using *custom sum* loss function 83
19 RF model performance comparison with different amount of features selected per tree, using *custom sum* custom loss function with weighted voting. 84
20 RF model performance comparison with different amount of features selected per tree, using variance reduction 84

21 RF performance comparison using reduced features (-6) 85
22 Hyperparameter summary for RF models using alternative custom loss function 92
23 Model predictions with test and train datasets 96

1 Introduction

The ATLAS experiment is one of the major experiments at CERN that records data produced at the Large Hadron Collider (LHC). The LHC is a powerful particle accelerator, in which two high-energy particle beams are accelerated close to the speed of light, before they are made to collide [1]. Upon colliding, the particles release an enormous energy, which results into creation of new particles. These particles do not live long, before decaying into different particles.

ATLAS (A Toroidal LHC ApparatuS) itself, is a massive detector, weighing 7,000 tonnes and having approximately 100 M readout channels. It is designed specifically to observe different properties of particles after their collision, recording properties such as trajectory, momentum and their energy [2].

One of the key points to be taken from these observations is the existence of missing transverse energy, which is inferred from an imbalance in recorded momentum. This means that certain particles are able to escape the detector, without being observed. One of such particles are neutrinos, whose existence we are aware of. However, it is possible that, upon colliding, an exotic particle, such as a dark matter particle, will be created.

The main difference between regular matter and dark matter is that it does not interact with the electromagnetic force, meaning that it does not emit light. In addition, similarly as neutrinos, they cannot be simply observed by a detector [3].

By using machine learning methods and simulated data, it is possible to create models that can differentiate between signal and background data. Briefly said, the signal data is data that can be of interest, which is generated using different hypotheses. The background data, on the other hand, is intended to simulate collisions resulting from known physics. After training a machine learning model on simulated data, with the intention of differentiating between the background and the signal data, it is possible that it would yield useful information regarding the actual data observed from the detector.

2 Background

2.1 Physics Behind

2.1.1 Standard model

The universe is made up of fundamental particles, whose interactions are described by four fundamental forces - electromagnetic, weak, strong and gravitational. The intention of the standard model is to encapsulate the interaction between these particles and forces. The standard model does not include gravity, however, since gravity is much weaker than the other forces, the standard model nevertheless has a great predictive power.

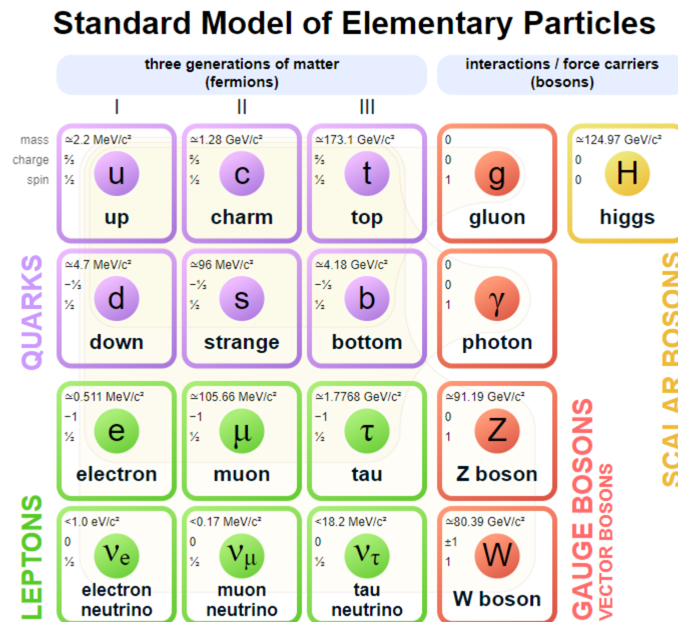


Figure 1: The Standard Model [4]

The standard model consists of seventeen fundamental particles, that can be further divided into two main groups: fermions and bosons. The fundamental difference between the two groups is that fermions act as the building blocks of matter, whereas bosons act as mediators of interaction [5]. Fermions can also be further separated into quarks and leptons.

Quarks
All the six quarks have a colour charge, hence they interact with the strong force. In addition, quarks also carry electric charge and weak isospin, meaning that they also interact with electromagnetic and weak forces.

Leptons
The important distinction between leptons and quarks is the forces they interact with. First of all, they do not carry a colour charge, hence they do not interact with the strong force. Secondly, three of the leptons are neutrinos, that also do not interact with the electromagnetic force. This means, that the only force they interact with is the weak nuclear force, making their detection difficult.

Gauge Bosons
The Gauge Bosons are also known as the force carrying particles that are able to interact with strong, weak and electromagnetic forces.

Higgs Boson
The Higgs boson is a scalar boson, discovered in the ATLAS experiment. It differs from the Gauge bosons by the fact that Higgs boson has a spin equal to 0.

Dark matter
Despite the standard model being the best explanation of the fundamental particles and their interactions so far, it leaves certain phenomena, such as dark matter, unexplained[5]. The key differences between regular matter and dark matter is that the dark matter does not emit light, and it does not interact with strong force, making its detection challenging. There are several hypotheses and types of evidence indicating the existence of the dark matter, one of the most known being observational evidence from galaxy rotation curves [3]. Dark matter is estimated to account for making approximately 27% of the universe, outweighing visible matter six to one. One of the key goals of experiments at CERN is to better understand and potentially to uncover the mystery surrounding the dark matter.

2.1.2 Jets

A high energy particle collision, such as the proton-proton collision at the LHC, has an ability to produce jets. A jet on its own is a collection of hadrons produced by hadronization, which occurs as the following [6]:

- Strongly interacting quarks or gluons are produced in a collision event. For simplicity, let us consider an electron-positron collision.
- One of the possible products of such a collision is a quark and anti-quark pair: $e^+ + e^- \to q + \bar{q}$, which will be assumed in this scenario.
- Upon reaching a certain separation distance, the strong interaction of quarks is strong enough to produce new quark-antiquark pairs.
- The quarks join together, producing a combination of hadrons that can be recorded in a detector. This process is known as hardronization.

A collection of hadrons with such a narrow spread is also known as a jet. Typically, two or more jets will be produced. In an electron-positron collision, one will typically be towards the direction of a quark and the other towards the direction of an antiquark, due to them having equal momentum [7]. However, in a proton-proton collider, the constituents of protons will carry some of their momentum. This means that upon their collision, they may contain different amounts of momentum, hence the jets will commonly be closer to each other. Moreover, more jets may be produced if a high energy gluon is produced in the process, in which case hadrons will be produced along it, as displayed in Figure 3.

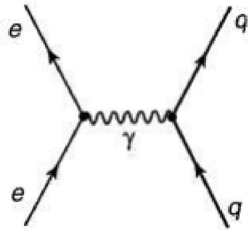


Figure 2: Electron and positron producing a quark, anti-quark pair [6]

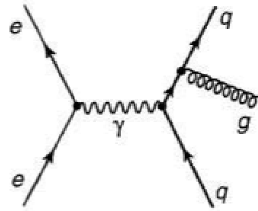


Figure 3: Electron and positron producing a quark and an anti-quark in addition to a gluon [6]

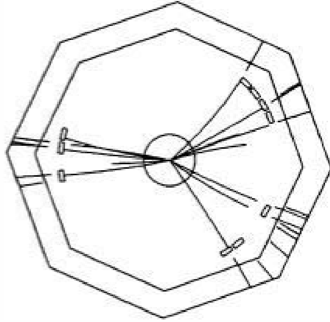


Figure 4: A typical three jet event visualized [6]

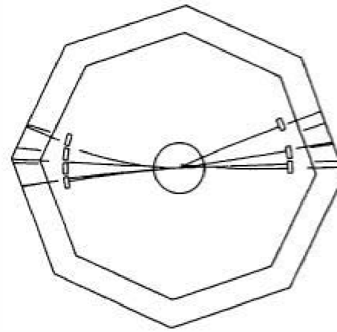


Figure 5: A typical two jet event visualized[6]

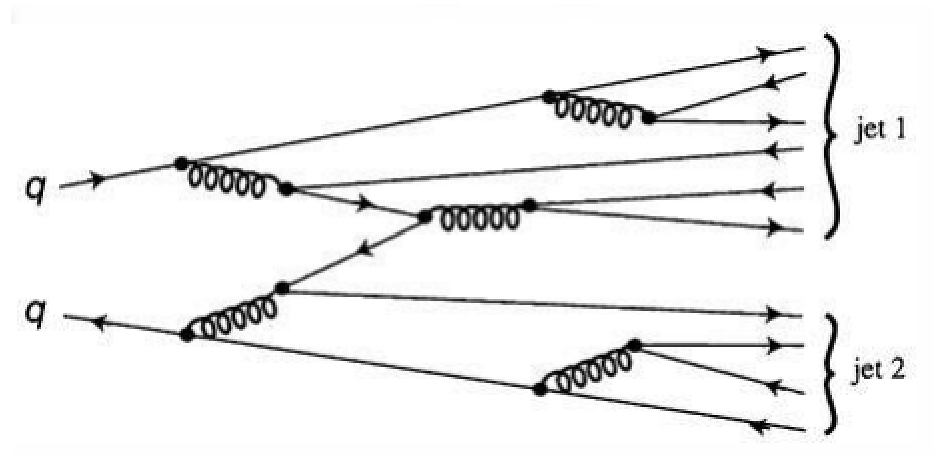


Figure 6: Hadronization and jet formation [6]

2.1.3 Taus

Tau particle is the most massive of leptons, having the mass of approximately 1.7768 GeV, making it the only lepton that can possibly decay into hadrons. A hadronically decaying tau has a detector signature that is similar to a quark or gluon jet. However, specialized tau reconstruction algorithms are capable of distinguishing them with an appreciable efficiency [8].

Tau Decay

Particle decay occurs when an unstable particle decays into several other particles. An important property of the particle decay is that the decayed particles will be less massive than the original particle, however the total energy must be conserved.

The typical tau decay is illustrated in the Figure 7. Approximately 65% of the time, the tau will decay hadronically. The rest of the time it decays into a tau neutrino, electron and electron neutrino or into a tau neutrino, muon and muon antineutrino. However, there are two main difficulties with these type of decays. The first one is the difficulty to determine that electron or muon indeed came from tau to begin with. The second one is that additional neutrinos, compared to hadronic decay, additionally reduces knowledge of tau's original energy and direction

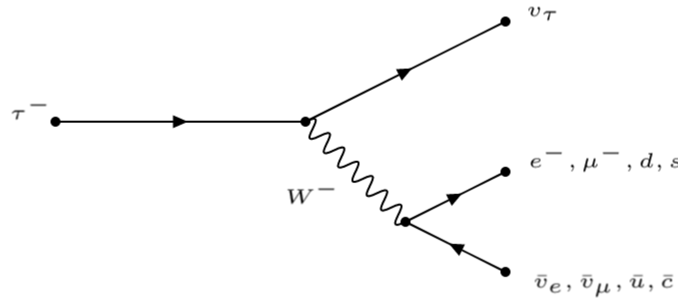


Figure 7: Tau Decay

2.2 Machine Learning and CERN

The particle collider experiments at CERN attempt to uncover the unresolved issues of the standard model and potentially discover new particles. Being able to differentiate between the hypothetical particles and known particles in simulated data by the use of in-depth analysis and machine learning methods is a key part of this process. At the moment, the most commonly used machine learning algorithms for this purpose are neural network and decision-tree based algorithms. In this thesis, the focus will be placed on random forest algorithm specifically.

3 Thesis Outline

3.1 Research questions

The main question that this thesis will attempt to answer is whether currently used ML models for separating background and signal data in high energy physics can be improved. Specifically, can a custom, specialized loss function improve the performance of a ML algorithms used in high energy physics? There is a wide variety of different, conventional loss functions optimizing regular ML models, based on different criteria. However, is it possible to develop a loss function specific to the purpose of separating signal and background data that outperforms the conventional ML methods?

In order to create an improved and specialized loss function, several properties of ML models will have to be considered. How are the models measured and compared? What kind of impact does a custom loss function have in terms of the results? Is it viable in practice? All of these questions will have to be considered when assessing whether a new loss function increases the chances of discovering new particles based on the data from the ATLAS experiment. To answer them, the performance of different ML models will have to be carefully compared and evaluated.

3.2 Research Methods

The most suitable research approach in this study is quantitative research. The goal of using a different, specialized loss function in a machine learning model is to directly improve its performance. Hence, upon training different ML models with different loss functions, it is possible to directly assess their performance based on selected performance measurements. The performance result is a collection of quantitative data that can be numerically compared between different models, allowing the aforementioned hypotheses to be tested.

In order to test whether a theoretical loss function is going to provide any improvements, it needs to be applied in practice. To achieve it, different ML models, using different loss functions will be implemented, trained, evaluated and tested on the same sets of data.

3.3 Plan For Evaluation

The evaluation of models' results will mainly be performed by comparing sensitivities of ML models using custom and conventional loss functions. Nonetheless, numerous steps must be taken into an account to ensure the integrity of the evaluation. Since different loss functions attempt to maximize or minimize different properties, it is likely that their performance is optimal using different hyperparameters. Hence, a thorough hyperparameter testing is necessary for all the loss functions tested and compared. Once the hyperparameter optimization is completed, the best performing models for each of the loss functions can be compared.

The goal of ML models developed and trained in this thesis is to achieve the highest sensitivity possible. Hence, sensitivity measures provide the necessary information regarding models' performance in order to make just conclusions. Other metrics, that do not take significant computational time, such as average predictions, can be of interest in the early stages of development, as they provide a general insight on the models' performance. However, they do not have a direct impact in making the conclusions regarding the models' final performance and results.

3.4 Thesis structure

The thesis will be divided into two core parts. The first part will focus on the relevant background concepts. To begin with, an explanation of the ATLAS experiment will be provided. This includes the goals of the experiment, how it is performed, and lastly the details of the ATLAS detector. This section will also contain the necessary information regarding the data that will be available in this thesis, including an explanation of datafiles and their features. Lastly, relevant machine learning concepts, such as decision trees and loss functions, in addition to theory behind the custom loss function will be provided.

The second part will mainly focus on the analysis of ML models. Different loss function will be applied to random forest and decision tree models, following an in-depth optimization and hyperparameter tuning. The main focus will be placed on assessing whether the custom loss function can provide better results, when compared to the conventional loss functions. Lastly, the findings will be discussed and presented.

3.5 Expectations

In the best case, the expected result for this thesis is to find a loss function that provides a noticeable improvement over loss functions currently in use for the background/signal separation. It is, however, impossible to guarantee. Nonetheless, throughout a thorough experimentation and analysis process, several hypotheses will be tested and documented. Despite whether the results improve or not, a difference in the performance from conventional loss functions is anticipated.

3.6 Relevant Research

Using custom loss functions in the field of HEP has been attempted before. Research done by Adam Elwood and Dirk Krücker, is one of such attempts, which has played a considerable inspiration to this thesis [9]. In their research, a custom loss function maximizing statistical significance has been developed, presenting noticeable improvements in the performance.

The fundamental difference between the custom loss function developed in this thesis, and in the research of Adam Elwood and Dirk Krücker is the underlying ML algorithm used. The custom loss function in this thesis utilizes the unique architectural properties of decision trees, whereas the loss function developed and applied in the aforementioned research is used in neural network algorithms. Despite that, a large inspiration has been taken from this research in terms of the general idea to maximize the statistical significance.

4 The ATLAS Experiment

4.1 LHC

Figure 8: The Aerial View of the Underground LHC [10]

The LHC (Large Hadron Collider) is a particle accelerator, having the length of 27 kilometres. As the name implies, the main purpose of LHC is to accelerate and collide particles. In particular, it accelerates beams made of bunches of protons with 10^{11} protons per bunch, close to the speed of light. The proton bunches collide with the approximate frequency of 30MHz, each of them containing up to 60 proton-proton collisions. This means that LHC can generate up to 1 billion particle collisions per second.

The LHC is made of superconducting magnets to boost the energy of the particles to be collided. The strong magnetic field provided by the superconducting magnets is necessary to guide the high-energy particles beams. In order to ensure that the electromagnets are superconducting, they are chilled to temperatures as low as 1.9K. Additionally, the pipes must be kept at an ultra-high vacuum, which assists the particle beams in avoiding collisions with the gas molecules inside the particle accelerator [1].

4.2 Particle detectors

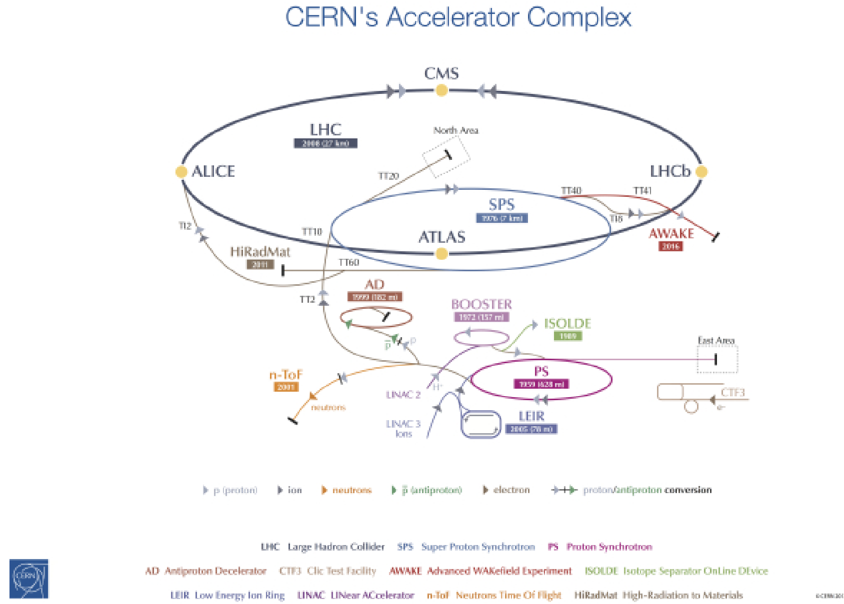


Figure 9: CERN's Accelerator Complex [11]

As seen in Figure 14 represented by yellow dots, there are multiple detectors at LHC, all of which are responsible for the capture of particle collisions. Despite each of the detectors being fit for their particular purpose, their underlying concept remains similar. The collisions occur within a detector, where each of the decay-produced particles interacts with the detector material, depositing their energy. By extracting information such as particles' charge, momentum, energy, direction and such, the type of decay-particles can be reconstructed. This information can in turn be used to reconstruct the particles before their decay.

4.3 The ATLAS detector

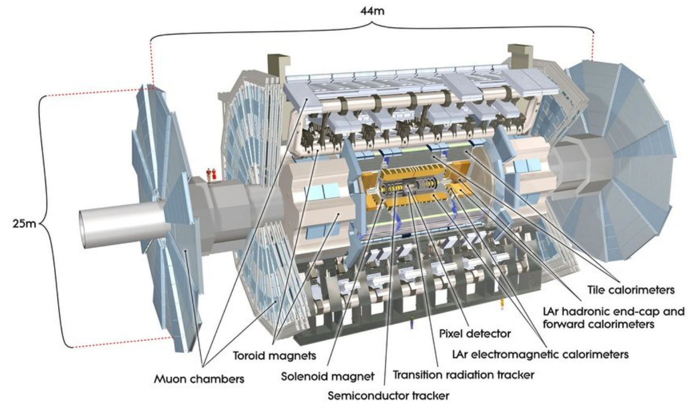


Figure 10: The ATLAS Detector [12]

The ATLAS detector is 46 meters long and has the diameter of 25 meters, in addition to weighting 7,000 tonnes. Although the ATLAS detector is complex and consists of thousands of different parts, its general physical construction can be divided into four key components: *the inner detector*, *the calorimeter*, *the magnet system* and *the muon spectrometer*, each playing a crucial role in extracting the information necessary to reconstruct the particles [2].

4.3.1 The Inner Detector

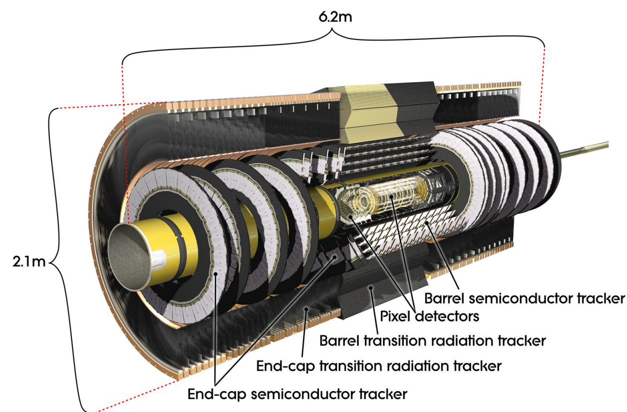


Figure 11: ATLAS Inner Detector [13]

The inner detector is a key component of the ATLAS detector, which allows the measurement of the direction, momentum, and charge of electrically-charged particles produced during the collisions. The inner detector is made of a pixel detector, a semiconductor tracker and a transition radiation tracker[14].

4.3.2 Calorimeter

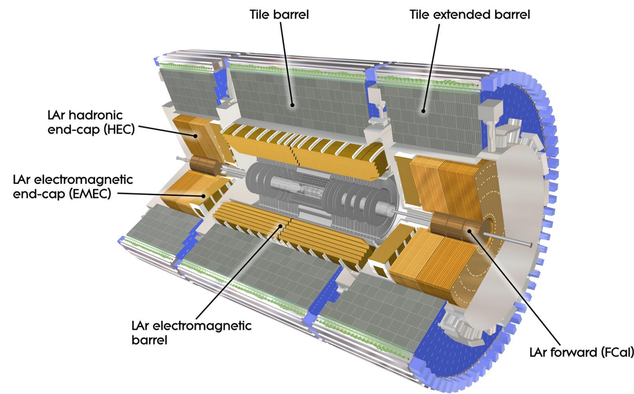


Figure 12: ATLAS Calorimeter [15]

The main function of a calorimeter is to measure the energy of particles as they pass through the detector. By using layers of absorbing, high-density materials that stop incoming particles, the calorimeters are able to absorb the incoming particles and force them to deposit their energy. The calorimeter's layers are also interleaved with *active* medium that measures their energy. The ATLAS detector uses electromagnetic and hadronic calorimeters, placed at the centre and at the end of the detector. The electromagnetic calorimeters measure the energy of electrons and photons, whereas the hadronic calorimeters measure the energy of hadrons.

Calorimeters are able to stop most particles, however not muons and neutrinos. Muons require different types of measurement, whereas neutrinos cannot be detected by the ATLAS detector and appear as missing transverse energy [16].

4.3.3 Muon Spectrometer

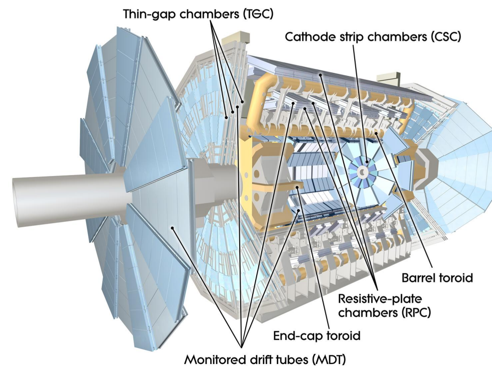


Figure 13: ATLAS Muon Spectrometer [17]

Since muons pass through the inner detector and the calorimeter without being stopped, there is a need for a different apparatus in order to measure their appearance. This task is performed by a muon spectrometer. The muon spectrometer consists of Thin Gap Chambers and Resistive Plate Chambers, responsible for triggering and 2nd coordinate measurement in the central region, Monitored Drift Tubes, which measure the curves of the tracks, and lastly Cathode Strip Chambers that measure precision coordinates at the end of the detector [18]. Additionally, the muon spectrometer refines the precision of momentum measurement.

4.3.4 Magnet System

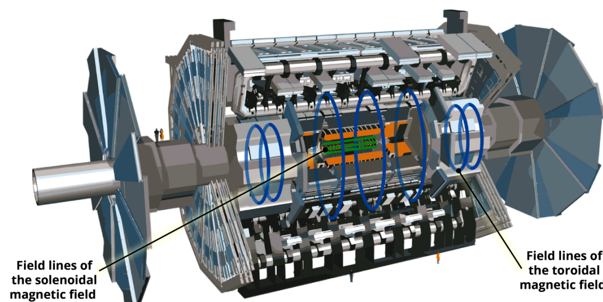


Figure 14: CERN's Accelerator Complex [19]

The magnets in the ATLAS detector are needed to bend the trajectories of charged particles, which allows the detector to measure their momentum and charge for further analysis. The magnet system consists of two superconducting magnets - solenoid magnet and toroid magnet.

The solenoid magnet surrounds the inner detector and is responsible for bending charged particles to measure their momentum. The toroid magnets, two of which are placed at the end-caps and one surrounding the centre of the experiment, are used to measure the momentum of muons.

4.3.5 Cross-Section

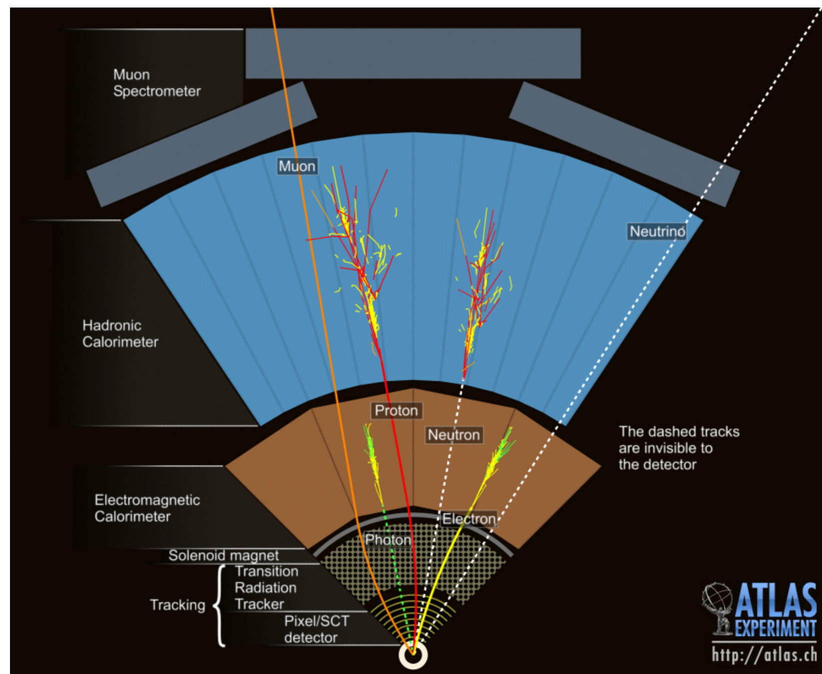


Figure 15: Event Cross-Section in a computer generated image of the ATLAS detector [20].

Figure 15 visualises ATLAS' capture of particles from the perspective of its cross-section. An electron will pass through an inner detector, leaving a track and finally stopping at the electromagnetic calorimeter. A photon will also leave a track at the electromagnetic calorimeter, however, it will not leave a track in the inner detector. Since protons are part of the hardonic particles, they will primarily interact with the hadronic calorimeter, however they also leave a track of their path in the preceding layers of the detector. Similarly, neutrons are also hadronic particles, hence they will also interact with the hadronic calorimeter, depositing their energy there. However, unlike protons, they will not leave a

track in the preceding layers. Muons will pass until the final layer of the detector, muon spectrometer, leaving a track behind. Lastly, the detector is unable to capture the presence of neutrinos. They leave the detector without leaving any tracks and appear as a part of the missing energy.

4.3.6 Trigger Systems

The vast majority of events from the particle collisions do not contain data of interest, that is, data that can lead to a discovery of new particles. Considering this, and the fact that there's approximately 10^9 interactions every second, a data selection is needed for an efficient processing [21]. The trigger system in ATLAS is designed to select data from the events that can be of interest.

The trigger system is split into two levels. The first level is hardware-base and reduces the data from 40MHz to 100kHz. When assessing what data is worth to be processed further, the trigger mainly considers certain parameters from the calorimeter, using a calorimeter trigger, and muon spectrometer, using a muon trigger. This process namely includes checking information regarding electrons, photons, τ-leptons, Jet/Energy-sum and additional data from the muon spectrometer [22].

The second level trigger is software-based, hence the decisions it makes are significantly slower, however it enables more complex assessments. If the criteria is met, the data is stored for offline analysis. At this point, the data has been reduced to approximately 1000Hz.

5 Data

The data, which will be used in this thesis, consists of simulated and real counterparts. The data itself contains background and signal samples. The background samples consist of particle decays from known physics, whereas the signal data consists of hypothesized exotic particle decay.

5.1 Simulated Data

The simulated data is necessary for numerous reasons. Unlike the real data, it does not come from the actual ATLAS detector. Instead, it is simulated using Monte Carlo simulations, based on current theoretical models. The simulation occurs in the following steps [23]:

Event Generation: The simulation process begins by generating the collision events. The final states of proton-proton collisions are generated based on on theoretical calculations, phenomenological models and experimental inputs.

Detector Simulation: Once the events have been generated, their interactions within the ATLAS detector are simulated.

Digitisation: In this step, the data is written into a format similar to that of the real output of the detector.

Reconstruction The same principles of reconstruction of the real data are applied to simulated data.

The key difference between the simulated and the real data, apart from the obvious of it being simulated, is that the simulated data contains labels, providing concrete information regarding the type of decays that have been simulated. This is the essence that allows the supervised machine learning models to be applied for this problem.

The main purpose of ML models in this problem is to separate signal and background data in the best way possible. In order to do so, the ML models must learn the difference between the patterns of background and signal data. In supervised machine learning, this requires data to be labelled, which the real data from the ATLAS detector is not.

In addition, simulated data allows an in depth analysis of the models. Not only is the signal and background data labelled in the simulated data, but also each of the constituent of background; that is, each type of decay from known physics. It provides the possibility of choosing the type of decays the background data should consist of, and the hypothesis chosen for the signal. Lastly, as new hypotheses arise regarding the exotic particles, it is possible to train the already implemented models on new data.

5.2 Real Data

The real data is the actual data as observed and reconstructed from the ATLAS detector. It contains the same features as simulated data, however it has no labels to follow along. Its main purpose in this thesis is to verify the integrity of simulated data, as described in Section 5.6.

5.3 Data files

The data files used for training, evaluating and testing ML models in this thesis consist of eleven background files and a single signal file. Each of the background files corresponds to a specific particle decay from known physics. The background files and their descriptions are summarized in Table 1.

File name	Description
diboson.MC16a.hdf5	Direct production of WW, WZ or ZZ
ttbar.MC16a.hdf5	Top and antitop quarks, produced as a pair
ttX.MC16a.hdf5	Top and antitop produced as a pair, and in addition either W, Z, photon or an extra pair of top and antitop
singletop.MC16a.hdf5	Top and antitop quarks
wenu.MC16a.hdf5	W boson, decayed into electron and its neutrino
wmunu.MC16a.hdf5	W boson, decayed into muon and its neutrino
wtaunu.MC16a.hdf5	W boson, decayed into tau and its neutrino
zee.MC16a.hdf5	Z boson, decayed into electron and positron
zmumu.MC16a.hdf5	Z boson, decayed into two muon leptons
znunu.MC16a.hdf5	Z boson, decayed into two neutrinos
ztautau.MC16a.hdf5	Z boson, decayed into two tau leptons

Table 1: Description of datafiles

5.4 Features

The available features in the dataset used in this thesis can be divided into three main parts: weights, meta-information and active features.

Weight features

In general, weight features are needed to match the number of simulated events to that of the real data. The difference in the amount of events is not accidental, and can typically be attributed to the cost of simulating data, or rare events being oversimulated, to get a better representation of possible variance. Nonetheless, in this dataset, the counts of simulated data are adjusted by using the following weights: *Lumiweight*, *mcEventWeight* and *pileupweight*. While weights directly impact the decision-making process of models used in this thesis, they are not part of the active features that the models learn from.

Meta-information

As implied, meta-information features contain information about the data itself. In this thesis, the feature *SampleID* has been used for data selection.

Active features

The active features include features that actively participate in the ML model training process. The features, along with their explanations, are summarized in the Table 2.

Feature name	Feature definition
tau_n	Numbers of taus in the event
jet_n	Numbers of jets in the event.
jet_n_btag	Numbers of jets with a b-quark associated
mu_n	Numbers of muons in the event
ele_n	Numbers of electrons in the event.
met	Missing Transverse Energy
met_phi	Orientation in radians of met in the XY-plane
ht	$\sum_i^{tau_n} tau_i_pt + \sum_j^{jet_n} jet_j_pt$
meff	met + ht
METoverPTMean	$\frac{MET}{ht/(jet_n+tau_n)}$
METSig	$\frac{MET}{\sqrt{\sum et}}$, where $\sum et$ is the scalar sum of energy in the transverse plane
object_pt	The transverse momentum of the physics object
object_phi	The phi angel of the physics object.
object_eta	Pseudorapidity of the physics object
object_width	$\Delta object_phi^2 + \Delta object_eta^2$
object_mtMet	$\sqrt{2 \cdot object_pt \cdot E_T^{miss} \cdot (1 - cos(object_delPhiMet))}$
tau_1_charge	Electrical charge of first tau
sumMT	$\begin{cases} tau_1_mtMet + tau_2_mtMet, & \text{if } tau_n \geq 2 \\ 0 & \text{otherwise} \end{cases}$

Table 2: List of active features

5.5 Data cuts

Certain data cuts have to be applied to the simulated data to ensure its validity. Cuts, as described in the Table 3, have been applied for this purpose. In particular, these cuts remove close to 100% of multi-jets, which is necessary, as they tend to be poorly modelled by the Monte-Carlo simulation.

jet_1_delPhiMet	>	0.4
jet_2_delPhiMet	>	0.4
jet_1_isBadTight	==	0
jet_2_isBadTight	==	0

Table 3: Data cuts applied in the ML analysis

5.6 Feature and simulated data validation

Simulated data is central in producing ML models. If there are severe discrepancies between real and simulated data, the resulting model, which is trained on the simulated data, may not provide adequate results when making predictions on the real data, despite its performance on simulated data. Hence, it is important to ensure that the simulated data is aligned with the real data and that it meets general expectations.

One way of validating the simulated data is by using feature validation. In this process, the distributions of different features of simulated and real data are plotted and compared. For a chosen feature, each individual background event is plotted as a stacked histogram, along with signal and real data. In an ideal scenario, the ratio between simulated and real data would be equal to one, meaning that they have the same distributions. It is, however, not always the case. In the plots below, the ratio of simulated to real data is also displayed as a separate figure, below histograms. In this case, the x-axis remains the same, displaying the feature values, whereas the y-axis displays the simulated to real data ratio.

This section will include an overview over the most important features, as shall be discussed in upcoming sections. The full list of feature validation figures is included in the Appendix B. In addition to the aforementioned data cuts, the feature validation plots contain two additional data cuts, as described in the Table 4. These cuts are necessary, to ensure that the data validation plots do not contain real data that can potentially contain signal, as to avoid any bias.

met	>	400 GeV
ht	>	1000 GeV

Table 4: Data cuts applied in feature validation plots

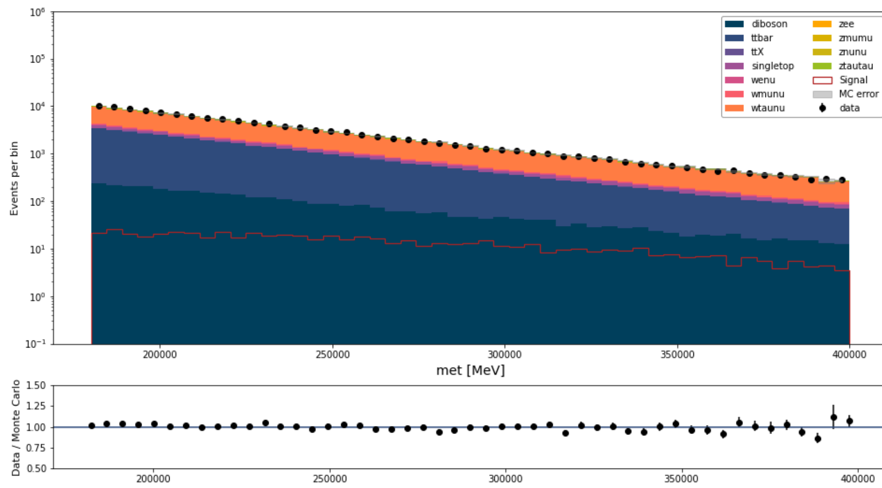


Figure 16: Feature validation for feature *met*

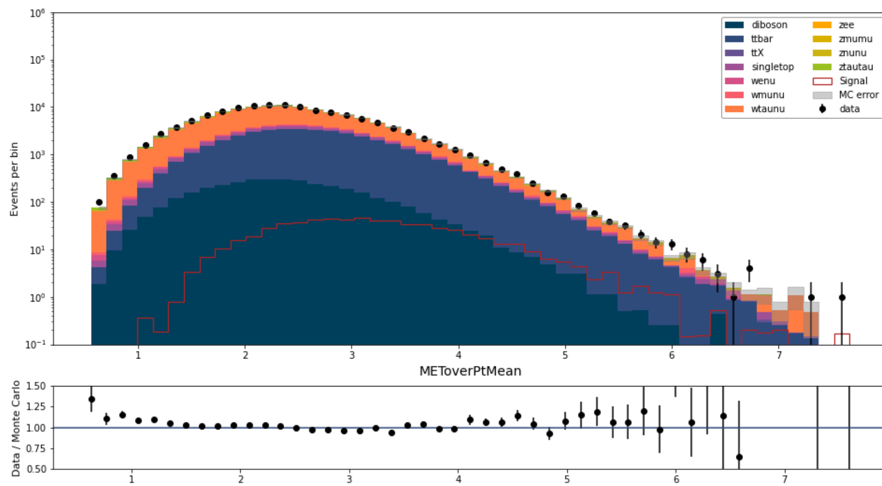


Figure 17: Feature validation for feature *METoverPtMean*

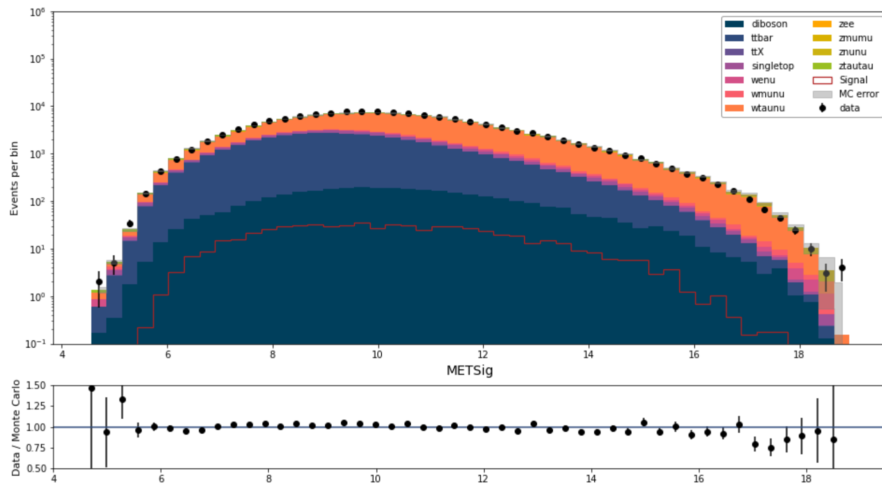


Figure 18: Feature validation for feature *METSig*

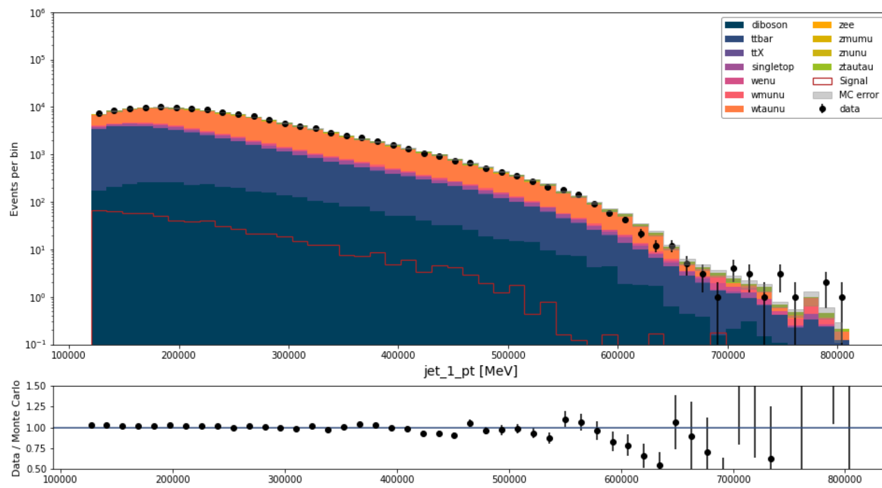


Figure 19: Feature validation for feature *jet_1_pt*

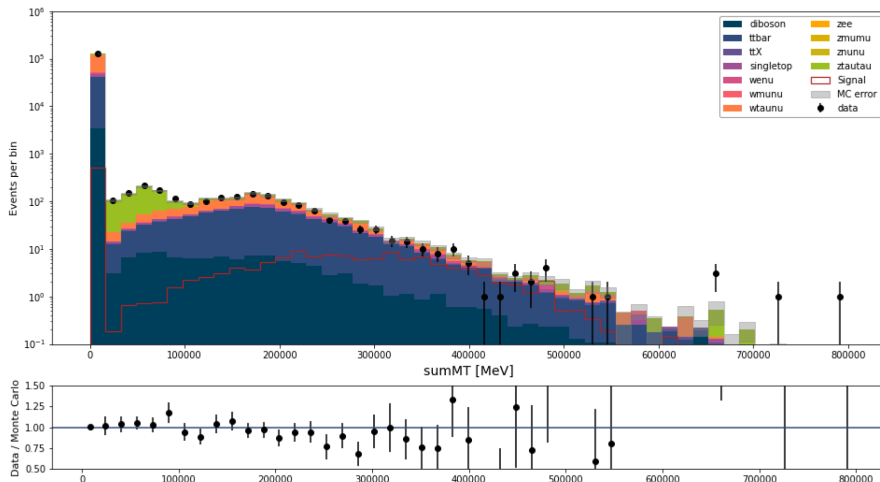


Figure 20: Feature validation for feature *sumMT*

As seen from figures above, the quality of simulated data varies between the features. The *met* feature has a satisfactory distribution. For the most part, its simulated to real data ratio remains at approximately 1. It does, however, have some deviations, where the ratio falls either above or below 1. The most prominent deviations appear towards the end of the figure, at the highest energy measurements.

While *jet_1_pt* feature has a good distribution in the early section, its quality falls off in with higher energy values. The amount of real data becomes significantly lesser than its simulated counterpart, starting from approximately $jet_1_pt > 550000$. As seen in Table 2, the variable *jet_1_pt* is a part of multiple feature calculations. As a result, it is anticipated that the distributions of the features related to *jet_1_pt* can be negatively impacted by its inconsistency with the real data.

Features *METoverPtMean* and *METSig* follow similar patterns. In both cases, their simulated to real data ratio is accurate in the midsection, however, the ratio falls off in the low and high ends of their respective minimum and maximum values.

Lastly, the feature *sumMT* is quite inconsistent with the real data. Up until approximately 190 000 MeV, its distribution is adequate. However, as the energy increases, the ratio consistently falls below 1. It is also one of the features where real to simulated data ratios have the highest errors.

In general, simulated data ratios are sufficiently adequate, to the extent where models, trained on simulated data, could be applied to real data as well.

6 Machine Learning

In traditional programming, the underlying logic and rules, responsible for the program's behaviour, are created by the person writing the program. In machine learning, on the other hand, the underlying algorithm is responsible for creating the rules defining the program's output, that are learned based on the data and its results. Machine learning can be further divided into two general categories, *supervised* machine learning and *unsupervised* machine learning.

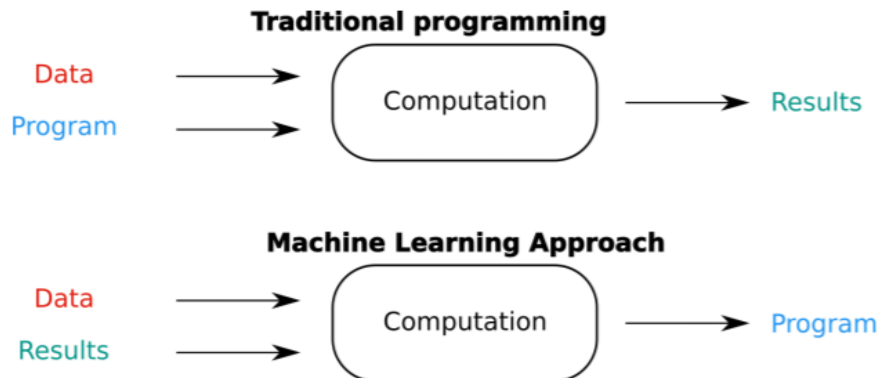


Figure 21: Machine learning vs Traditional programming [24]

Supervised machine learning
Supervised machine learning is defined by its use of labels - the ground-truth values for data. As a ML model is being trained, and data is "fed" into it, the labels are used to assess whether the model performs well. Supervised learning includes a variety of ML algorithms, including neural networks, decision-tree based algorithms, support vector machines and more.

Unsupervised machine learning
Unlike supervised ML algorithms, unsupervised ML algorithms do not use data labels. Instead, they attempt to discover the hidden patterns in the data, utilizing data grouping. Different clustering algorithms, such as k-means clustering and probabilistic clustering, and neural networks are examples of some of the commonly used unsupervised ML algorithms [25].

6.1 Decision Trees

Decision tree learning is a commonly used machine learning algorithm, which, as implied, is based on the idea of decision trees. A simple binary decision tree consists of a parent node, left and right children. The parent node is split according to a certain criterion that maximizes a given loss function, such as information gain for classification problems or variance reduction for regression problems.

Binary decision tree building algorithms work by recursively splitting the data into two subsets, based on the features available in the dataset. The unique values of each of the feature act as thresholds, based on which the splits are made. For categorial features, the dataset is split according to whether the current data entry's feature value is equivalent to that of selected threshold's. For continuos features, on the other hand, the data is split according to whether the current data entry's feature value is higher or lower than that of selected threshold's. Nonetheless, the resulting subsets are used to calculate the chosen loss function, and the feature maximizing it is chosen as a tree branch. The parent node is then split into two subsets, that become nodes of their own, for which the process is repeated until an end condition is met. The end condition typically is either a maximum depth achieved, a parent node not containing enough data samples, or the chosen loss function not improving any more. At this point, the node becomes a leaf node, which has an output and is able to make a prediction. For regression problems, the output is the average value of the dependent variable. For classification, the output is the majority vote of dependent variables within the node.

6.1.1 Variance reduction

The most commonly used loss function for regression decision tree building is variance reduction. The calculation of maximum variance reduction in a single node works as follows:

- A feature and its threshold value are selected.
- The entire dataset of a parent node is split into two subsets, based on whether the feature is lesser or greater than the selected threshold.
- The variance for both of the children and the parent node is calculated using the following formula:

$$\sum_{i=1}^{n} W_i(X_i - \overline{X}) \ , \tag{1}$$

where:

$$W_i = \text{The weight of the current data entry}$$
$$X_i = \text{The current data entry}$$
$$\overline{X} = \text{The weighted mean of the current data entry}$$

- Lastly, the variances of left and right children nodes are added, and the resulting value is subtracted from the parent node, which results in variance reduction score.
- The process is repeated for all the thresholds and all the features until the maximum variance reduction is found.

6.1.2 Custom loss function

Motivation
Random Forest (RF) machine learning models, with the intention of separating complex signal and background data, typically use variance reduction loss function as its basis. The process of using these models goes as follows:

1. A regression RF model is created and trained on selected data.
2. The model predicts evaluation or test data, resulting in predictions ranging from 0.0 to 1.0, 0.0 indicating that the data is likely of background type and 1.0 indicating that the data is likely of signal type.
3. A threshold is chosen for what to classify as signal and background.
4. Using this threshold, the regression model is used as a classification model, predicting everything below the threshold as background and everything above as signal.
5. Sensitivity values are then calculated for each of the threshold chosen. The threshold providing the highest sensitivity is then selected to represent the model.

The issue with this method and also the reasoning for the need of a custom, specialized loss function is that the desired goal is not optimized directly. In this case, variance reduction attempts to perfectly separate background and signal data. However, this is not feasible in practice, due to the problem's inherit complexity. Therefore, sensitivity is used as the main metric to assess model's performance. Sensitivity is calculated as $\frac{s}{\sqrt{s+b}}$, where s is the signal, correctly predicted as signal and b is background, incorrectly predicted as signal. Briefly said, sensitivity tells the likelihood of finding signal data. The idea behind developing a custom loss function is to maximize the sensitivity directly, which would also provide the model with a possibility to be used as a classification model without any workarounds.

Another drawback of variance reduction in HEP-related problems, is that it places an equal emphasis to background, incorrectly classified as signal, and signal, incorrectly classified as background, when assessing which split is to be made. This does not accurately reflect the desired goal, as a model able to provide clear regions of signal, at the expense of some of it being classified as background is much preferred over a model that classifies most of the signal correctly, at the expense of increase in background misclassified as signal.

Theory

The intention behind the custom loss function is to separate signal and background data as well as possible, while addressing the aforementioned problems with variance reduction. Since the main priority of this loss function is to achieve the least background misclassified as signal, while still providing a reasonable overall separation, it is unlikely that it will provide a more homogeneous separation than variance reduction, which focuses entirely on it. However, if the custom loss function is able to reduce false positives at an expense of an increase in false negatives, it could still provide a better performance classifying regions of signal.

The ability to separate signal from background is quantified by the significance, approximated by the expression: $\frac{s}{\sqrt{s+b}}$, which is also the core idea behind the custom loss function. The hypothesis is that, if it is maximized for each of the splits made, the machine learning model would not only be able to separate background and signal, but also reduce the amount of background misclassified as signal.

Implementation-vise, the theory behind the custom split is to maximize the sensitivity in the right child nodes. Whenever a split is going to be made, the increase in the sensitivity of the right child will be calculated and the split with the highest increase will be chosen. In theory, this would result into the rightmost nodes being most signal-like and leftmost being most background-like. The idea can be visualized in the following figure:

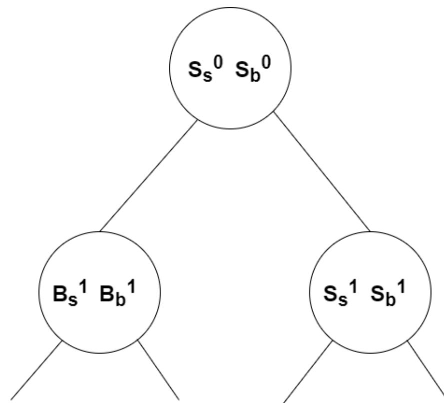


Figure 22: Decision tree node split visualized

, where:

S_s^0 = Number of signal entries in the root node
S_s^0 = Number of background entries in the root node
S_s^i = Number of signal entries in the signal node after split i
S_b^i = Number of background entries in the signal node after split i
B_s^i = Number of signal entries in the background node after split i
B_b^i = Number of background entries in the background node after split i

The custom loss function is then calculated in two ways. The first one, from here on referred to as *custom sum* loss function, is calculated as:

$$\frac{\sum_k S_s^k}{\sqrt{\sum_k S_s^k + \sum_k S_b^k}} \tag{2}$$

The second one, from here on referred to as *custom non-sum* loss function, is calculated as:

$$\frac{S_s^k}{\sqrt{S_s^k + S_b^k}} \tag{3}$$

The key difference between these loss functions is the *custom sum* loss function taking into a consideration the intermediate nodes in its calculation. The *custom non-sum* loss function, much alike other, traditional decision tree loss functions takes into a consideration only the parent and its children nodes.

6.2 Random forest

Decision trees excel in runtime performance, and their implementation is relatively uncomplicated. Despite that, it is unlikely for a decision tree algorithm to provide adequate results for complex tasks on its own. However, decision trees are also a basis of other, powerful machine learning algorithms that have a potential to provide improved results for complex problems, one of such algorithms being the Random Forest algorithm. Random forest is an ensemble algorithm mainly based on the idea of ***Bagging***.

Bagging, as a general idea in machine learning, refers to a variety of algorithms that follow the principle of converting multiple *weak* learners into a single, *strong* one. In the context of machine learning, a *weak* learner is a machine learning model that does not provide adequate results, however, it is also one that surpasses a random guess. By creating multiple weak learners and combining them together, the resulting learner has a potential to be a strong one.

One of the extensions of this idea, applied specifically to decision trees, is known as random forest. In this case, a *weak* learner is a tree that only takes into a consideration a few of the total features to be used in its learning process. Different trees are trained with different samples of features and data. An ensemble of these trees is also known as Random forest [26].

The random forest algorithm works as follows:

I For each estimator, a tree is built:

  i A random set of features based on *max_features* hyperparameter is chosen.

  ii Data with replacement is sampled from the training data.

  iii A decision tree is trained according to its parameters.

II A prediction is made:

  i Each of the decision tree makes a prediction on the data.

  ii The average value of the combined tree outputs is the output of the random forest.

6.3 Implementation

Libraries, commonly used for decision tree algorithms, are heavily optimized to provide the best performance. At its expense, the ability to customize them, such as by implementing a custom split criteria, tends to be limited. Therefore, in order to test the hypothesis, the decision tree building algorithm was built without the use of high level ML libraries. The tree building algorithm is based on two main functions: one for building the tree itself (*build_tree*) and one for finding the best split(*get_best_split*).

Since the loss function is directly dependent on the counts of signal and background entries, their weights have a direct impact on the implementation. When calculating the amount of signal or background entries within a node, their weighted counts are used, rather than the count of entries themselves. This provides a more accurate representation of the data, based on which the model learns its patterns.

```
1  def build_tree(self, dataset, curr_depth=0, s_s =0, s_b =0):
2    num_samples = dataset['summed_weight'].sum()
3    best_split = {}
4
5    if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
6      best_split, s_s, s_b =self.get_best_split(dataset, num_samples, s_s, s_b)
7
8      if best_split:
9        if best_split["var_red"]>self.min_split_gain:
10         right_subtree = self.build_tree(best_split["dataset_right"],
11                                         curr_depth+1, s_s, s_b)
12         left_subtree = self.build_tree(best_split["dataset_left"],
13                                         curr_depth+1, s_s, s_b)
14
15         return Node(...)
16
17   leaf_value = _get_avg(dataset['data_type'], dataset['data_weights'])
18   return Node(value=leaf_value)
```

The *get_best_split* function starts by checking whether the stop conditions are met. These conditions include the weighted minimum samples in the current split and the current tree depth. The model checks that the current depth and minimum samples in a node are not higher than that, which has been specified in the model's parameters. If the stop conditions have been met, the current dataset belongs to a leaf node. Therefore, the weighted mean is calculated by the function *get_avg*, which calculates the following:

$$\frac{\sum_i data[i] * data_weights[i]}{\sum_i data_weights[i]}, \tag{4}$$

where:

$$data[i] = \text{Data type of current data entry (0 for background, 1 for signal)}$$
$$data_weights[i] = \text{The product of all the weights belonging to the current data entry}$$

If the stop conditions were not met, the function continues by recursively finding new best splits at an increasing depth. By calling *best_split* function, it receives a dictionary containing information about the split, including the feature chosen, left and right splits, improvement in the loss function and more. If the dictionary is empty, it means that the model, after iterating through all features and thresholds selected, was not able to find any improvements and thereby the stop condition has been met again. If an improvement was found, however, the model continues the process recursively for the left and the right children that were provided by the *best_split* dictionary.

```
1  def get_best_split(self, dataset, num_samples, s_s, s_b):
2
3    best_split = {}
4    new_s_s, curr_s_s = 0, 0
5    new_s_b, curr_s_b = 0, 0
6    max_all_feature_split_measure = -float("inf")
7
8    for feature in self.features:
9      feature_values = dataset[feature]
10     all_thresholds = self.get_reduced_threshold_list(np.sort(all_thresholds))
11
12       for threshold in all_thresholds:
13         dataset_left, dataset_right = self.split(dataset, feature, threshold)
14         if len(dataset_left)>0 and len(dataset_right)>0:
15           p, p_weights     = dataset['data_type'], dataset['data_weights'],
16           r_c, r_c_weights = dataset_right['data_type'], dataset_right['data_weights']
17           l_c, l_c_weights = dataset_left['data_type'], dataset_left['data_weights']
18
19           current_feature_measure, curr_s_s, curr_s_b = self.custom_split_func(...)
20
21           if current_feature_measure > max_all_feature_split_measure:
22             best_split["feature"] = feature
23             best_split["threshold"] = threshold
24             best_split["dataset_left"] = dataset_left
25             best_split["dataset_right"] = dataset_right
26             best_split["var_red"] = current_feature_measure
27             max_all_feature_split_measure = current_feature_measure
28             new_s_s = curr_s_s
29             new_s_b = curr_s_b
30
31   return best_split, new_s_s, new_s_b
```

As its name implies, *get_best_split* function is responsible for finding the best possible split. The function iteratively loops through all the features provided. For each of the feature, the function extracts the possible values that a feature may have, which then are used for the threshold selection. Due to performance constraints, checking all the thresholds for all the features was deemed too time-consuming. Therefore, a selected number of thresholds was chosen incrementally increasing from the lowest to the highest value of the feature by the use of the function *get_reduced_threshold_list*. The implications of this constraint are discussed in the Section 6.4.

Nonetheless, for each of the threshold, the dataset is split into two. The left split contains all the data whose selected feature value is less than or equal

to the threshold, and the right split contains the data with the feature value higher than the threshold. The function then checks if there's at least one data entry in each node. If so, the split measure is calculated. If it is higher than the current one, the *best_split* dictionary is updated with the new information. Upon iterating through all the features and all the thresholds, the information regarding the best split is returned.

```
1  @jit(nopython= True, fastmath=True, parallel= True)
2  def _custom_split(parent, parent_weights, child, child_weights, s_s, s_b):
3
4    child_background_count_sum = s_b
5    child_signal_count_sum = s_s
6    weighted_signal_count, weighted_background_count = 0, 0
7
8    for i in range(len(child)):
9      if child[i] == 1:
10       weighted_signal_count += child_weights[i]
11     else:
12       weighted_background_count += child_weights[i]
13
14   child_signal_count_sum += weighted_signal_count
15   child_background_count_sum += weighted_background_count
16
17   measure_child =  child_signal_count_sum / math.sqrt(child_signal_count_sum +
18                                                      child_background_count_sum)
19   measure_parent = s_s / math.sqrt( s_s + s_b)
20
21   measure = measure_child - measure_parent
22
23   return measure, child_signal_count_sum, child_background_count_sum
```

The last tree building function used is the one responsible for determining which split is the best one. The function *custom_split* takes in the parent node and its weights, the right child node and its weights, and lastly the total current sum of signal and background count in the right nodes. The sum of signal and background is updated based on the right child node. The measure is then calculated by the equation (2), which is returned along with the updated signal and background counts.

```
1  def make_prediction(self, row, tree):
2    if tree.value!=None: return tree.value
3    feature_val = row[tree.feature]
4
5    if feature_val<=tree.threshold:
6      return self.make_prediction(row, tree.left)
7    else:
8      return self.make_prediction(row, tree.right)
9
10 def predict(self, data):
11   predictions = [self.make_prediction(row, self.root) for (_,row) in data.iterrows()]
12   return predictions
```

Lastly, the function *predict* takes in a dataset and is responsible for generating predictions for each of the dataset's entry. For each row in the dataset,

it calls the function *make_prediction*. The function *make_prediction* utilizes the tree that has been built and stored internally upon calling the *build_tree* function. The initial *tree* value is equal to the root node. This means that *tree.left* is equivalent to its left child, and likewise *tree.right* is its right child. Since, in this implementation, only the leaf nodes have an output value, if the current node does not have one, a prediction cannot be made using in. In such case, the process continues by checking the feature and its value, based on which the current node has been split on. If the feature value of the current data entry, whose output value is being predicted, is less than or equal to the feature value in the current tree node, the process is repeated with its left split, else the process is repeated with its right split. Once a leaf node is found, the leaf value, which is its weighted mean, is returned.

As for a short example, if a root node has found that it maximizes its loss function with the following splits: $MET <= 200000$ and $MET > 200000$ for left and right children respectively, and the MET value of the data entry/row that is being predicted is 190000, the predict function will iterate to its left child and then check the conditions of *tree.left*. The process would then be repeated for the feature based on which the split has been made in *tree.left*, and so forth until a leaf node has been reached.

```
1    def get_sensitivity_values(data, preds):
2      data['prediction'] = np.array(preds)
3      signal_data = data.loc[data['data_type'] == 1]
4      background_data = data.loc[data['data_type'] == 0]
5      thresholds = np.arange(0.0, 1, 0.025)
6      .
7      .
8      .
9      for threshold in thresholds:
10
11       signal_predicted_as_signal =
12           signal_data.loc[signal_data["prediction"] > threshold]['data_weight'].sum()
13
14       signal_predicted_as_background =
15           signal_data.loc[signal_data["prediction"] <= threshold]['data_weight'].sum()
16
17       background_predicted_as_background =
18           background_data.loc[background_data["prediction"] <= threshold]
19               ['data_weight'].sum()
20       background_predicted_as_signal =
21           background_data.loc[background_data["prediction"] > threshold]
22               ['data_weight'].sum()
23
24       sensitivity =
25           signal_predicted_as_signal / math.sqrt(signal_predicted_as_signal + background_predicted_as_signal)
26
27       sensitivity_measures.append(sensitivity)
28
29       if sensitivity > max_sensitivity:
30         max_sensitivity = sensitivity
31         max_sensitivity_threshold = threshold
32
33     return thresholds, sensitivity_measures, max_sensitivity, max_sensitivity_threshold
```

An important measure to consider when assessing the model's performance is its sensitivity. The sensitivity measures are calculated by the function *get_sensitivity_values*. The function takes in data and their predictions generated by the aforementioned *make_predictions* function. The data is then separated to signal and background respectively. For each of the threshold selected, ranging [0,1] with increments of 0.025, signal predicted as signal, signal predicted as background, background predicted as signal and background predicted as background is calculated. In this case, a threshold is merely a value, which defines what data is defined as signal and what data is defined as background for classification purposes. Signal predicted as signal and signal predicted as background are calculated by counting the weighted amount of data entries in the signal dataset, whose predictions are respectively higher or lower than the threshold chosen. Background predicted as signal and background predicted as background are calculated the same way. The sensitivity is then calculated as seen in the lines $25-26$ in the code above. Additionally, the signal and background counts using the threshold value providing the highest sensitivity are kept a track of. When plotting sensitivities, the x-axis represents the thresholds and the y-axis represents the sensitivity value the model achieved with a specific threshold.

6.4 Implementation and performance assessment

Considering that the decision tree and random forest algorithms have been implemented without the use of verified machine learning libraries, ensuring their performance was essential. In order to do so, several techniques have been employed.

Comparing decision tree models

The model consistency between custom and verified implementations has been performed by training and comparing models implemented with *Sklearn*[1] library and models using a custom implementation. The models were trained and tested on the same sets of data, and their performance and feature selection was compared. The models also had the same parameters, however, the implementations are not identical. *Sklearn* library, to which the custom implementation has been compared to, is a heavily optimized, state-of-the art machine learning library, hence, despite the core algorithm being the same in both of the implementations, *sklearn*'s implementation has certain improvements over the custom implementation. Multiple constraints have been implemented in the custom version, in order to provide better runtime performance, which is anticipated to have a slight negative impact on the model's performance.

As seen in Figure 23 and Figure 24, the decision tree performance between different implementations supports the initial hypothesis that the custom implementation will have a disadvantage in terms of final results. Despite both models having the same parameters and using the same datasets, the custom implementation had a peak sensitivity value of 1.749, whereas the *sklearn's* implementation had the peak sensitivity value of 1.817. The sensitivity plots are also alike, and any inconsistencies between the two can be explained with the decision tree graphs.

The decision trees graphs, displayed in Figure 25 and Figure 26, display the features chosen by each of the model and the threshold they were split at. With the exception of one feature, all the features chosen during the splits were the same. Likewise, the thresholds for the selected features were also similar, and aligned with expectations. The slight inconsistencies can be explained due to aforementioned constrain issue with the custom explanation, or *sklearn* library taking a different approach to weights in variance reduction calculation.

[1] One of the most common python ML libraries, also known as scikit-learn https://scikit-learn.org/stable/

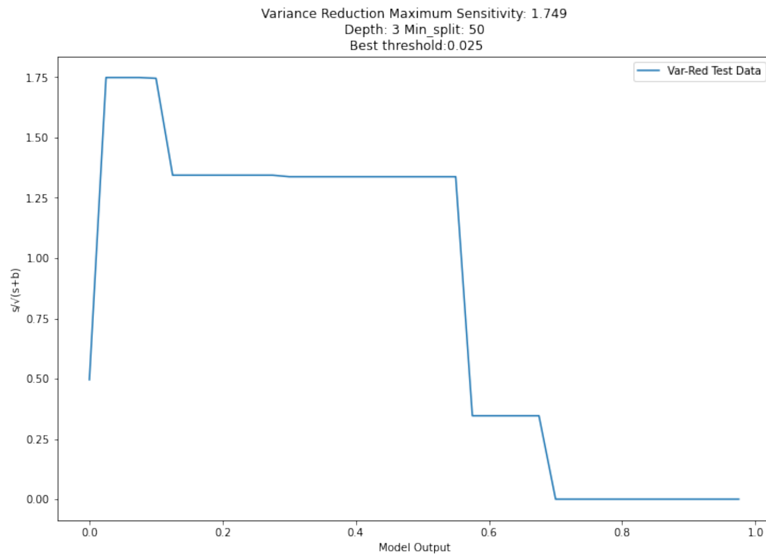


Figure 23: Sensitivity plot of decision tree model, implemented using a custom implementation

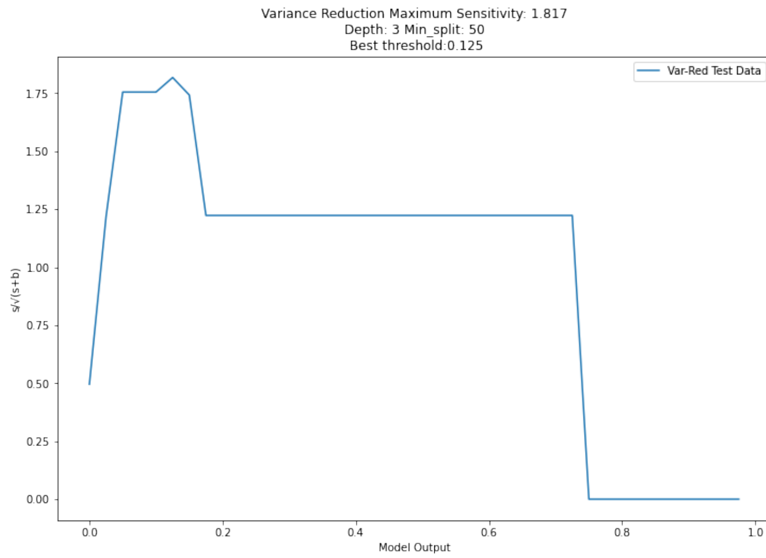


Figure 24: Sensitivity plot of decision tree model, implemented using *sklearn* library

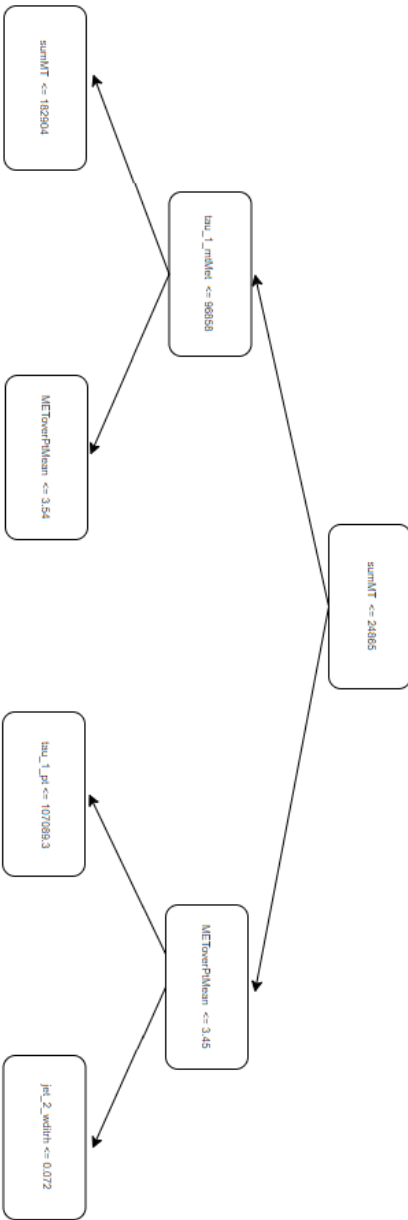


Figure 25: Decision tree graph of decision tree model, implemented using custom implementation

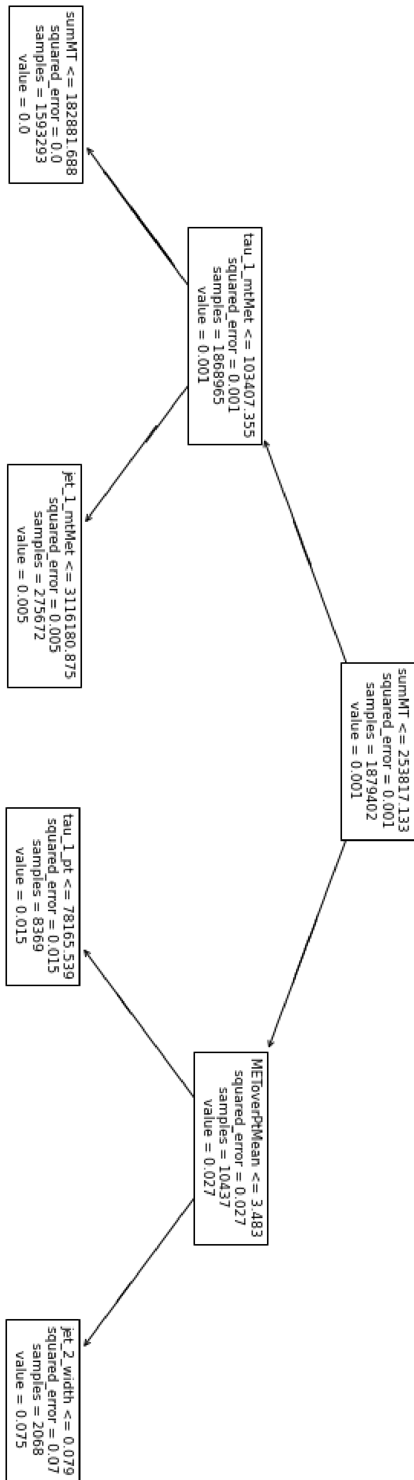


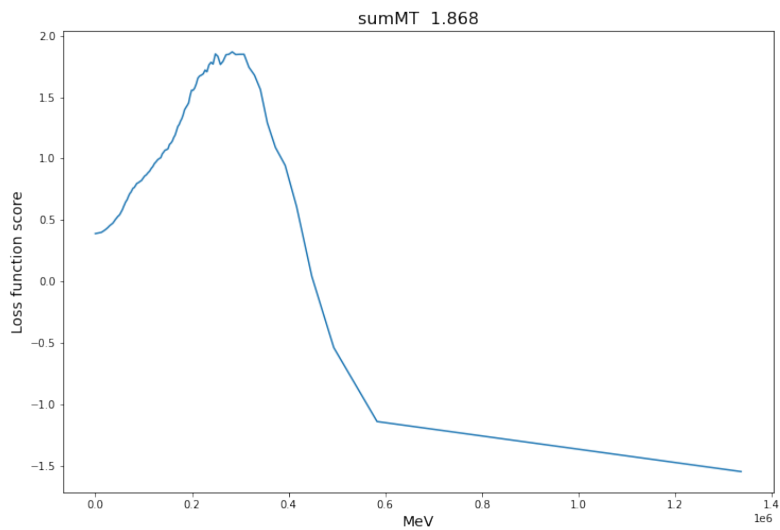
Figure 26: Decision tree graph of decision tree model, implemented using *sklearn* library

Custom implementation assessment remarks
Custom implementation comparison to *sklearn's* performance serves a purpose as a high-level overview check. The custom implementation is not intended to compete with *sklearn's* implementation, as that is not the point of this thesis. This comparison mainly provided an insight to the custom implementation, and increases confidence that there are no significant issues within it, in terms of performance achieved.

Verifying custom split choice
The verification of splits by the custom loss function has been performed by manually splitting the data for each selected threshold into children nodes, calculating the sensitivities achieved, and plotting the results. In this case, the parent node was the entire training dataset, just as it would be for the root node of a tree. In this process, just like in the tree building process, it has been split into left and right children based on feature and its threshold. The splitting process has been performed for each of the feature and its threshold, and their sensitivity values have been calculated.

The Figure 27 below shows an example of the aforementioned process' results, comparing two features that were able to achieve the highest, and second-highest loss function values - *sumMT* and *tau_1_mtMet*. The y-axis represents the custom loss function value, that is, the sensitivity of the right child node, subtracted by the sensitivity of its parent node. The x-axis, on the other hand, displays the particular threshold chosen. The title of individual figures also includes the feature chosen, in addition to its sensitivity increase, at its best performing threshold.

This test has been performed for two first splits, as made by the decision tree building algorithm. The features selected, and the increases were found to be matching the manual selection. However, this particular test was not only useful for the split verification. It also provided an insight on the impact reducing the amount of thresholds has on the final loss function value achieved. Testing every unique value for each of the feature was neither feasible nor practical due to time and performance constraints. As an example, the feature that is typically chosen by decision trees for its first split, *sumMT*, has over 55000 unique values. As a result, n threshold values have been selected incrementally from lowest to highest for each of the feature. Such constraint is anticipated to have a slight impact on the final performance.



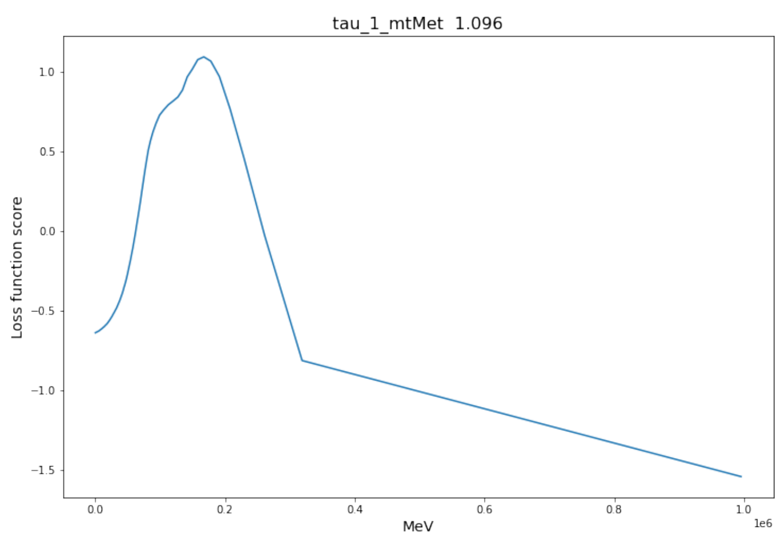


Figure 27: Comparison of top 2 features with the highest sensitivity increase achieved

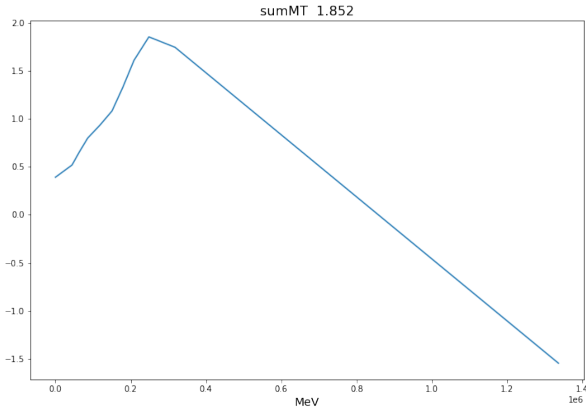


Figure 28: Threshold reduction performance, *n = 10*

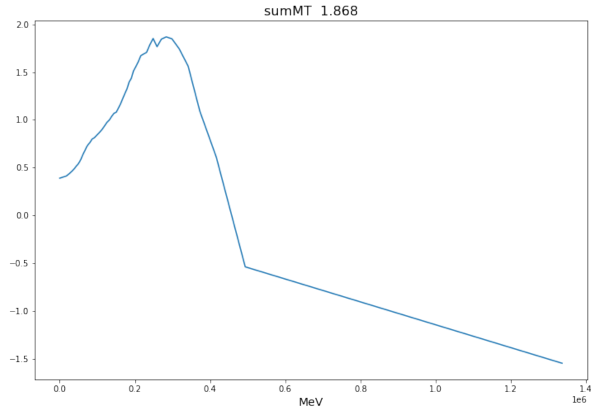


Figure 29: Threshold reduction performance, *n = 50*

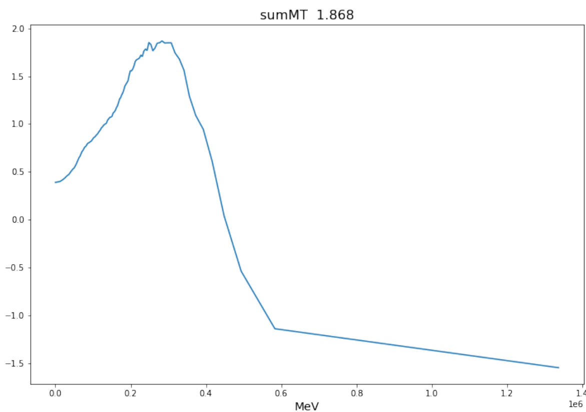


Figure 30: Threshold reduction performance, *n = 100*

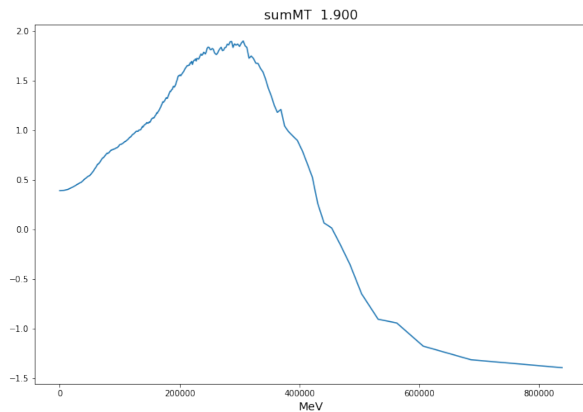


Figure 31: Threshold reduction performance, *n = 300*

The figures above display the differences in plots and peak results achieved by testing n selected thresholds for a selected feature. The absolute difference in the peak sensitivities achieved was not observed to be significant, and it did not make an impact in the feature selected. However, it is still theoretically possible that, when using a reduced amount of thresholds, a different feature than the best one may be chosen. In this case, the difference in the performance achieved between the two features would have to be minimal, or the feature plots, as seen above, would have to have a sharp peak that could be skipped with fewer thresholds. The first case is possible, however, that would also mean

that the features provide more or just about the same amount of improvement. The second case is also possible, yet, no such occurrence was observed in this test.

As seen in the Appendix C, which displays the figures for all the active features tested in custom split verification testing, many of the features follow a similar pattern. The performance increases with each threshold until a peak sensitivity is achieved, followed by a rapid decline. As an attempt to utilize this fact, and to minimize the negative impact of reduced the amount of thresholds, an alternative threshold selection approach has been tested. Using this approach, the threshold would be tested starting at its minimum, and the loss function value of n subsequent thresholds would be kept a track of. If n loss function values have been consistently decreasing, the peak loss function value before the decrease would be chosen for the split.

One of the issues with this particular method of threshold selection was the fact that, when using a large quantity of thresholds, the increase towards the maximum was not as steady as it seemed with a lower amount of thresholds, visualized by the Figure 32.

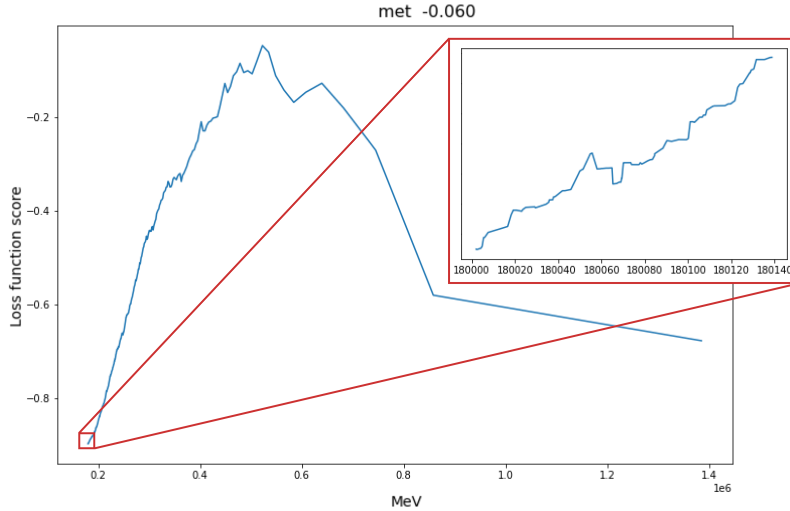


Figure 32: Close-up of loss function values calculated for thresholds

This obstacle could possibly be solved by setting a minium amount that the loss function calculation must decrease for it to count, however, it was also observed that a few features do not follow this pattern, especially as the tree depth increases. For example, the loss function values for the feature *jet_1_mtMet*, as displayed in the Figure 33, falls steadily for lower thresholds, followed by a rapid increase.

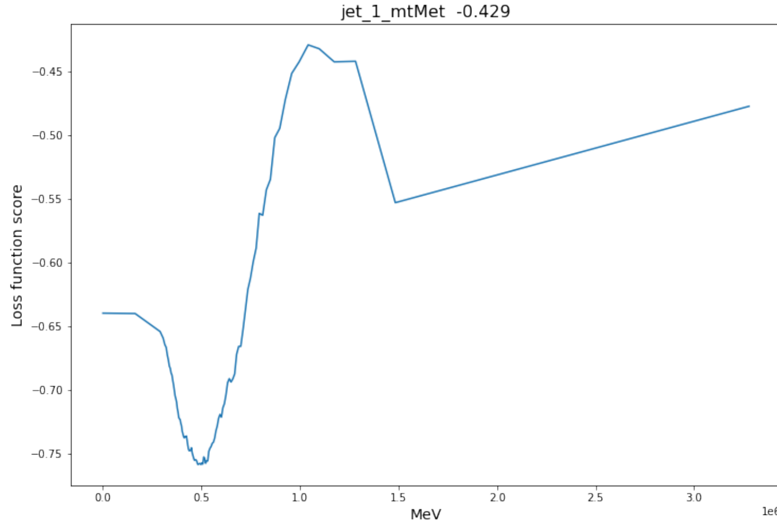


Figure 33: Loss function value plot for feature *jet_1_mtMet*, $n = 100$

6.5 Development Tools

A variety of tools has been used throughout the entire model development process. Some of which are typical to building machine learning models, others not as much.

The main development platform has been a combination of Google Colab and personal computer. Google Colab offers cloud computing, allowing for an efficient execution of code. While being sufficient in the earlier iterations of the project, its downsides shortly became apparent. The main downside being its inability to guarantee access to a GPU, which eventually became a necessity to perform thorough testing.

Despite the custom implementation using a GPU for the run time improvements, the efficiency was not on a par to the state-of-the-art python ML libraries. Building a new ML library was not the intention of this project. However, building models, whose run-time was sufficiently optimized, was a prerequisite for an efficient and thorough experimentation and analysis. Nonetheless, this meant that models were, to a reasonable extent given the circumstances, optimized performance-wise. However, their runtime was nevertheless not sufficient for google colab. As a result, there was an attempt to use a different cloud computing platform. The platform in mention is SWAN, provided by CERN. However, it was to no avail. The main issue with it was the difficulty of finding library compatibility with the pre-installed libraries. Hence, heavy computation has been left to be done by the use of a personal machine, whereas less intensive tests could still be performed by the use of google colab.

The aforementioned library, which allows GPU utilization for python, is called *Numba*. It allows translating python code into a highly-efficient machine code at runtime, which in turn allows it to utilize GPU. It does, unfortunately, come with its own drawbacks. The main drawback is the limitation of supported data types. In order for python code to be translated to an efficient machine code, it needs to use low-level data types. The *Numba* library supports the following data types:

- **Integers** up to 64 bits
- **Booleans**
- **Real numbers** single-precision (32-bit) and double-precision (64-bit)
- **Complex numbers** single-precision (2x32-bit) and double-precision (2x64-bit) complex numbers
- **Datetimes and timestamps**
- **Character sequences** (but no operations are available on them)
- **Structured Scalars** structured scalars made of the types above and arrays of the types above

While Python is likely the most commonly-used programming language used for machine learning, it is not the only one. However, despite the implementation not being limited to it, no satisfactory alternatives were found. Majority of alternative ML libraries based on different programming languages still do not support the custom implementation of loss functions, or they suffer from other drawbacks. An attempt was made to use R programming language that did, in fact, have a library supporting custom loss function implementation for decision trees. However, it had issues with efficiently handling files of *hdf5* format. In addition, it also had other drawbacks, such as comparatively slow data processing. Considering this, and the fact that decision tree and random forest algorithms are well documented, a decision has been made to use Python without the use of high level ML libraries.

7 Decision Tree Analysis

7.1 Hyperparameter testing and model optimization

Despite the fact that it is not anticipated for decision tree models to perform well on their own, a preliminary hyperparameter testing provides a general overview over the loss functions' performance. In addition, the computational time of single decision tree is much lesser than that of random forest, or other tree-based algorithms, allowing a relatively time-efficient testing.

7.1.1 Process

The hyperparameter testing has been performed with equal splits of training and evaluation datasets, each corresponding to randomly sampled data, equivalent to 25% of the total data and consisting of mixed background and signal entries. It should be noted that this particular test is intended to provide a high-level overview over the loss functions and not to make concrete conclusions regarding the performance of loss functions tested. For this reason, no test dataset was used to test the models that performed best on the evaluation dataset.

Nonetheless, the hyperparameter test included testing a combination of depths of [5, 10, 15] and minimum samples split of [2, 20, 50, 100]. For each of the hyperparameters, three regression models, representing each loss function tested, have been created and trained using the same parameters. The first loss function is the control loss function - variance reduction, as defined in Equation (1). Its purpose is to provide a baseline for custom loss function performance comparison. The remaining loss functions are the custom loss functions, defined by Equation (2) and Equation (3).

Upon training a model and getting their predictions for evaluation dataset, sensitivities have been calculated and plotted as described in Section 6.3. An example of such plot can be seen in the Figure 34.

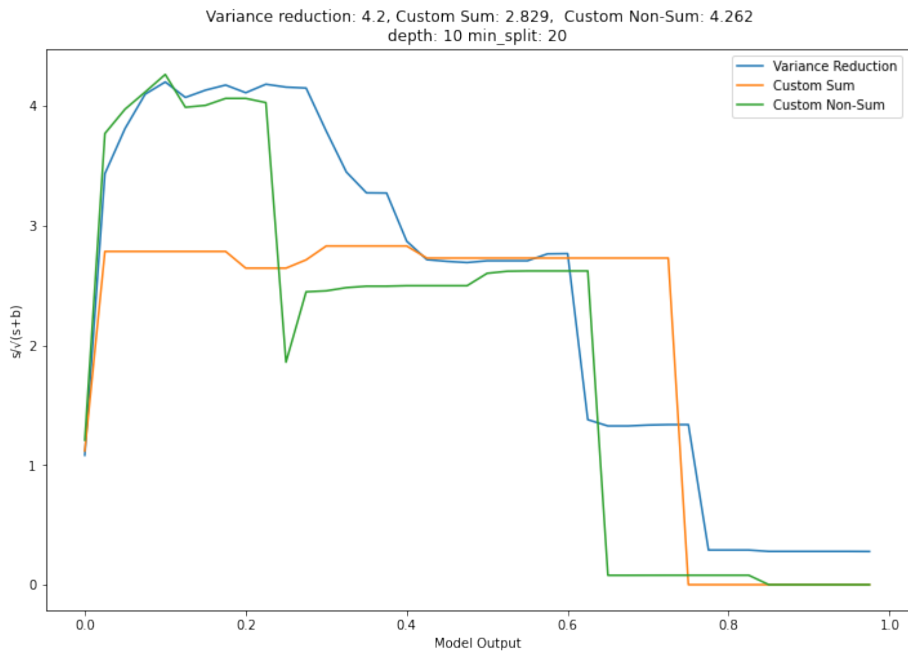


Figure 34: Sensitivity plots using different loss functions

7.1.2 Results

The results have been summarized in the tables below. For individual sensitivity plots, refer to Appendix A.

depth\min_split	**2**	**20**	**50**	**100**
5	**4.26**	4.23	**4.26**	4.16
10	3.68	4.20	4.24	4.25
15	2.68	4.06	3.94	4.13

Table 5: Peak sensitivities of variance reduction

depth\min_split	**2**	**20**	**50**	**100**
5	3.17	2.90	3.00	2.67
10	**3.02**	2.89	3.00	2.68
15	3.01	2.80	3.00	2.67

Table 6: Peak sensitivities of custom sum loss function

depth\min_split	**2**	**20**	**50**	**100**
5	4.05	4.07	4.06	3.97
10	4.19	**4.26**	**4.26**	4.23
15	3.94	4.16	4.23	4.22

Table 7: Peak sensitivities of custom non-sum loss function

The initial results were not highly promising for the custom loss function. The *non-sum* custom loss function had an on-par performance with variance reduction, resulting into the peak sensitivity of 4.26. The *sum* version, however, resulted into a comparatively poor performance, having the peak sensitivity of 3.02. Apart from it, the results were in align with the expectations. It was anticipated for the models having high depth and low minimum samples split to underperform, as are prone to overfitting. Likewise, models with low depth and high minimum samples split, were not expected to have the best performance, as they are more likely to underfit the data.

7.2 Observations

Splitting the data with the *sum* custom loss function does indeed provide relatively strong signal nodes in the right splits. However, it suffers from a major drawback of classifying a large portion of the data with a few splits. If the model only takes into a consideration the right splits and maximize their sensitivity, it quickly becomes unable to find improvements in the left splits. This would not be an issue if the model would reach the point at which the left-most nodes would not contain a significant amount of signal. In practice, this does not occur. As a result, a significant portion of signal is left at the same node as a large portion of background, which cannot be further split using these criteria.

An example of a decision tree using the custom *sum* loss function can be seen in Figure 35, which visualizes its key issues. The decision tree model in this example is slightly different from the regular models. Instead of using all the background files, only a single background file, *ztautau.MC16a.hdf5*, and

a single signal file, $G_1100_968_901_835.MC16a.hdf5$, have been used. This task is severely less complicated to solve for a ML model. The full background dataset consists of 11 separate background files of varying sizes, meaning that the signal to background ratio is typically significantly lower. Otherwise, the process for the split is the same as it would be for the entire dataset, hence the general patterns in the tree itself are similar. The end conditions for the decision tree algorithm were: *Minimum samples split: 20, Maximum tree depth: 3*. The depth of the tree is low and likely results into underfitting, however the purpose of this decision tree is to visualize the split process rather than efficiently separate background and signal files.

The first line of the node shows the feature and its value, on which the split, maximizing its loss function, has been made, followed by a "?" and the value of loss function for the split. If the value of a feature is less than or equal to the selected threshold, the data is separated to the left child, else to the right child. Each of the node also shows the intermediate count of weighted (having $w_$ as a prefix) and unweighted background classified as background (denoted as b_b), and signal classified as signal (denoted as s_s) counts. Lastly, the output of each of the leaf nodes is the weighted mean of *data_type* variable within them. The *data_type* is the dependent variable of this model. Its value 0 represents that the data is of background type and 1 that the data is of signal type.

The first noticeable issue in the tree displayed in the Figure 35 is that, despite the tree algorithm being allowed to reach the depth of 3, on the second left split, it is unable to find improvements and stops the split before reaching its maximum depth. As a result, the left-most node contains 176 signal entries and 2657 background entries. This corresponds to approximately 42 percent of the signal and 85 percent of the background. This isn't an issue inherently, as the main priority of the ML model is to get pure signal. Hence, if it is able to classify the remaining signal with a high accuracy, the model could still perform well. In practice, however, the impact of this issue is too great and is heavily reflected on the model's performance. It is also important to take into a consideration that the optimal tree depth, maximizing the performance of the model for this task, typically lies between 5 and 10. Hence, if the model fails to find improvements after merely a couple of splits, it is likely going to underfit the model. Apart from these issues, it can be seen that the theory itself works to an extent. The right splits do indeed maximize their sensitivity, as seen by high loss function values in the example decision tree below.

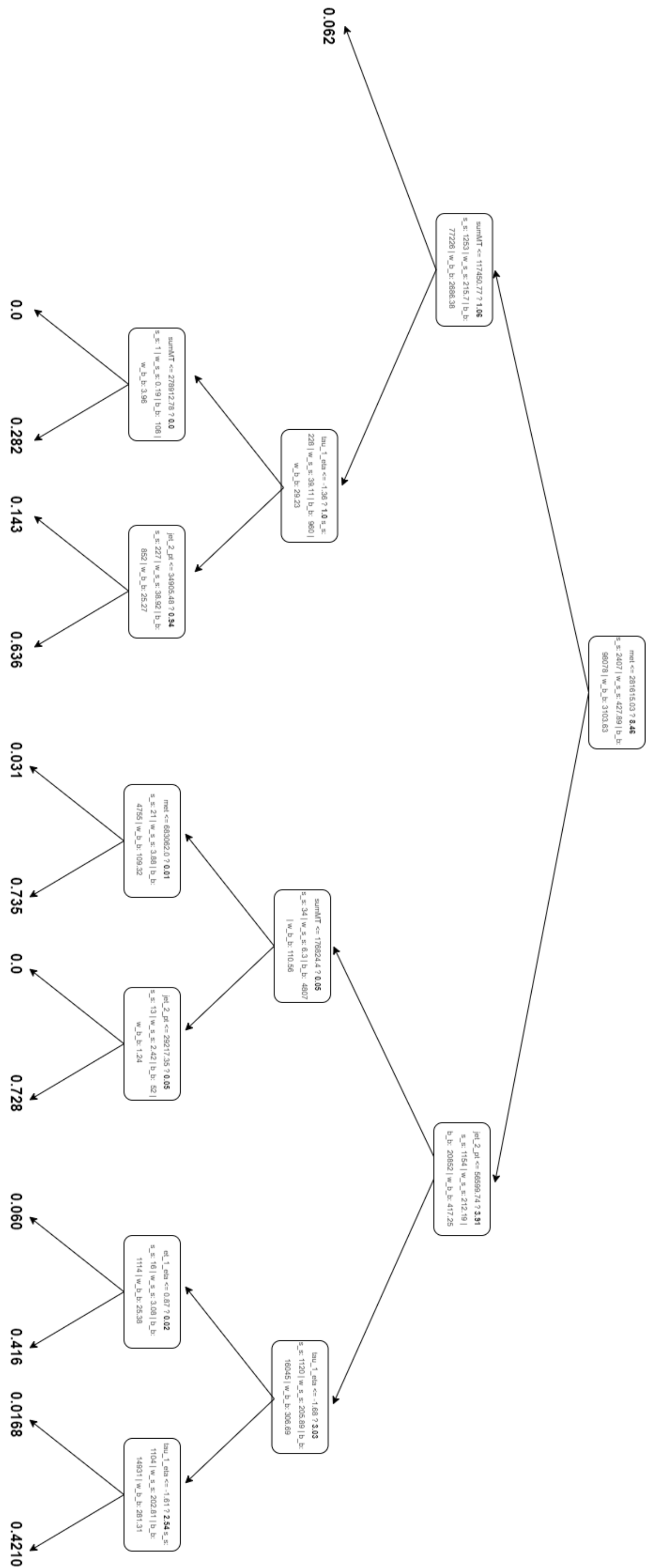


Figure 35: Example of a decision tree graph using custom *sum* loss function

8 Random Forest Analysis

8.1 Random Forest Using Classification

8.1.1 Theory

In Section 6.1.2 it was mentioned that one of the possible advantages of custom loss function is its ability to be used directly in classification models. This approach is closer to the nature of the custom loss function, as it already classifies right leaf nodes as signal nodes, and left as background in its decision-making process. In this case, the output of the leaves will no longer be the weighted average of the *data_type* parameter, which indicates whether a particular data entry is a background or signal. Instead of the prediction output being in the range $[0, 1]$, the output will be the either 0 or 1. If the majority of data in a tree leaf is of signal type, the output will be equal to 1 and likewise, if the majority is background, the output will be equal to 0. The performance comparisons will still be performed with variance reduction regression models using thresholds for classification purposes, as it provides better results than a regular RF classification model using information gain.

8.1.2 Process

Initial testing was performed to test the general viability of using RF models as binary classification models, without placing an emphasis on getting definitive conclusions regarding the general custom loss function's performance. In depth hyperparameter testing will be conducted in later sections. Nonetheless, the viability of using RF models as classification models directly has been performed by comparing classification RF models using custom *sum* loss function and regression models using variance reduction.

Note that, the reason for the variance reduction models being regression models initially, and only then being used as classification models, is because it generally provides better results in terms of peak sensitivity achieved. By selecting different thresholds for signal and background classifications, the precise threshold maximizing sensitivity can be chosen to represent the model. This is a common approach for maximizing sensitivity in HEP, that is not limited to random forest models.

8.1.3 Results

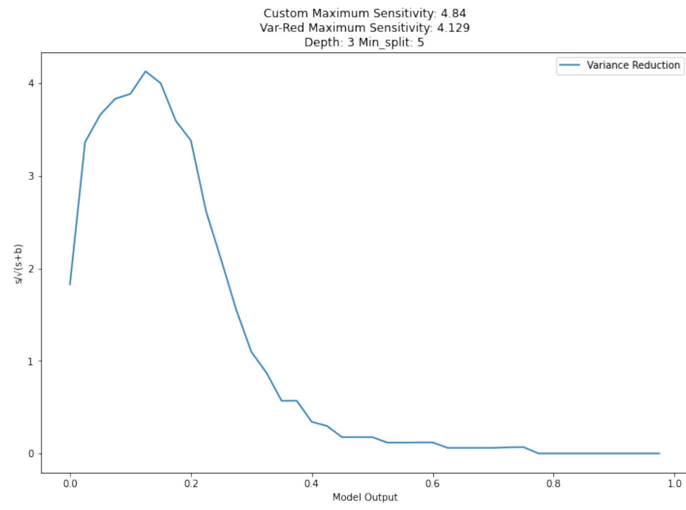
The figures below display the following information:

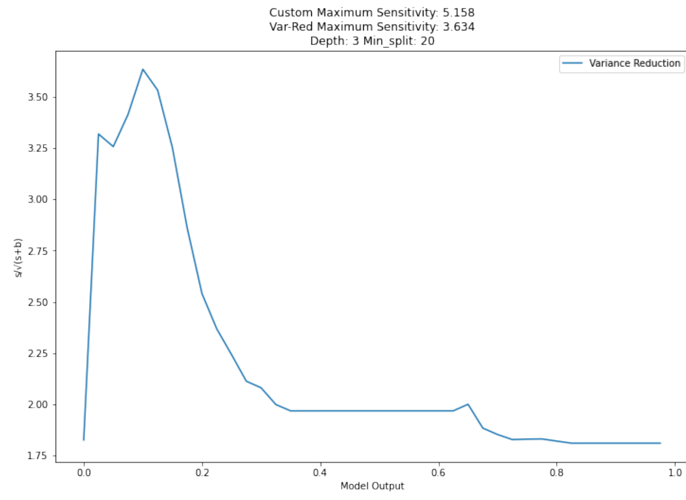
- Graphs for the weighted variance reduction. The x-axis represents the selected threshold, and the y-axis represents the sensitivity achieved using it.
- Maximum sensitivity achieved with RF regression model using weighted variance reduction (displayed in the title).
- Maximum sensitivity achieved with RF classification model using the custom *sum* loss function (displayed in the title).
- The *maximum depth* and *minimum samples split* parameters of the model (displayed in the title).

In addition, the counts of weighted signal classified as signal and background classified as signal were computed for both of the loss functions. In case of variance reduction, the threshold providing the highest sensitivity was selected.

Lastly, the RF model had the following parameters:

- Each tree considered 50, linearly increasing, unique thresholds from each of the feature when assessing which feature should be selected for the split.
- 30 total features were used. Each individual tree was assigned 6 randomly chosen features.
- Each tree was trained with 50% of the total training data.
- There were 15 total decision trees in the assembly of RF.
- Maximum depths of [3,6] and minimum samples split of [5, 20, 100] were tested.
- The sum-version of custom loss function was used.
- The models were tested on 50% of the total data.





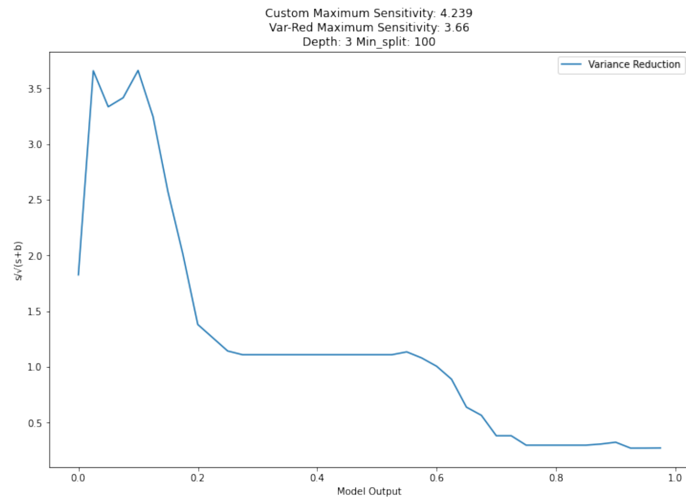
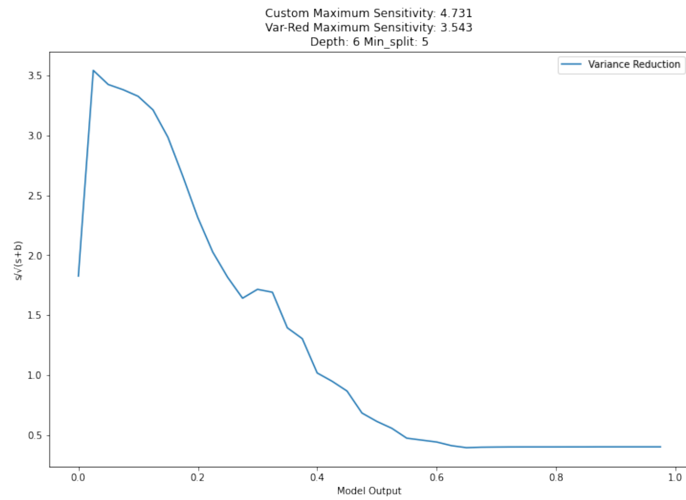
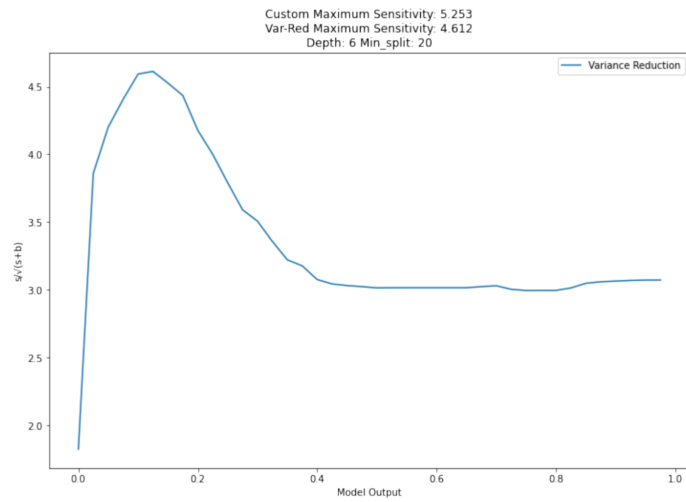


Figure 36: Sensitivity comparison of RF models using *custom sum* and variance reduction loss functions for different minimum splits, using depth 3





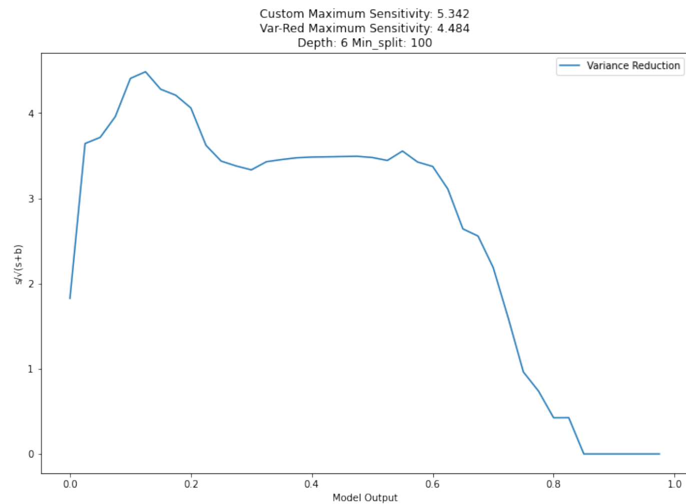


Figure 37: Sensitivity comparison of RF models using *custom sum* and variance reduction loss functions for different minimum splits, using depth 6

The results are summarized in Table 8. The table compares the maximum sensitivity (*max_sens*), weighted signal, correctly classified as signal (*w_s_s*) and background, incorrectly classified as signal (*b_s_s*) using the best performing models for different loss functions respectively. The difference in the sensitivities of models using custom loss function and variance reduction was approximately 0.731, with the custom loss function providing a better performance. The difference is substantial, however, the tests performed were not in depth, and additional analysis must be performed to make trustworthy conclusions regarding custom loss function's performance. Nonetheless, this test did confirm the ability of using custom *sum* loss function for classification models directly, which was the main intention of this test.

	w_s_s	**b_s_s**	**max_sens**
Custom loss function	136	772	5.342
Weighted variance reduction	103	400	4.612

Table 8: High-level performance comparison between custom loss function and variance reduction

Note that, an ensemble of 15 trees is generally considered low, and RF models are expected to provide significantly better results with more estimators. However, the intention of this test was to evaluate the general viability of RF models being used as a classification models with the custom loss function. In-depth hyperparameter testing will be performed in upcoming sections.

Hyperparameter testing has also been performed using the of the non-sum custom loss function with a classification model. However, it provided underwhelming results. The peak sensitivity remained at approximately 1.5, with close to no responsiveness to the changes in parameters. In comparison, a regression model using the custom loss function and the same parameters, provided maximum sensitivities of approximately 5.0.

8.2 RF performance fluctuation testing

8.2.1 The need for consistency

Previous testing indicates that the custom loss function has a potential to outperform traditional loss functions in achieving higher peak sensitivities. However, in order to achieve reliable conclusions, a thorough hyperparameter testing is necessary. For the hyperparameter testing to be meaningful, there is a need for consistent results. The deviations in models' performance may occur for several reasons, most of which can be attributed to the inherent randomness of *random* forest, in addition to workarounds used for performance improvements.

Since random forest uses a random selection of features and dataset samples for its trees, its performance can vary significantly if an insufficient amount of

trees has been used within a forest. The best way to minimize this risk is to increase the number of trees. The more trees a random forest has, the less likely it is going to be made of majority weak trees due to its randomness. If a low amount of trees is combined with a high amount of features, large deviations in performance may occur.

In the previous section, an ensemble of 15 trees has been used, each of them selecting 6 random features out of 30 available. Considering the low amount of trees and high amount of features, the model was a likely candidate for the aforementioned problem.

8.2.2 Intermediate results

To test RF fluctuations, multiple RF models with the same parameters have been created. Despite having the same parameters, train and test datasets, the models chose different, random set of features and dataset samples for their individual trees. Figures below display information for the performance fluctuations. The y-axis represents the peak sensitivity of each of the model and the x-axis represents *nth* model.

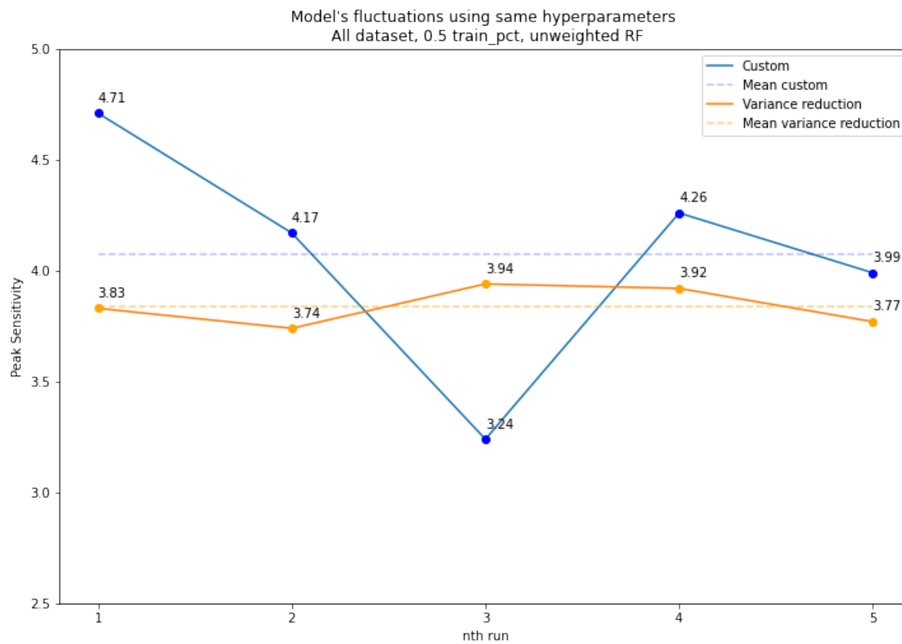


Figure 38: Deviations of peak sensitivities for RF models using same parameters

	Variance Reduction	Custom *Sum* Loss Function
Run 1	3.83	4.71
Run 2	3.74	4.17
Run 3	3.94	3.24
Run 4	3.92	4.26
Run 5	3.77	3.99

Table 9: Deviations of peak sensitivities for RF models using same parameters

Five runs were sufficient to ensure that a significant deviation exists between each of the model's results. The maximum sensitivities achieved by the model using custom loss function varied between 3.25 and 4.71. This difference is substantial enough for the custom loss function to go from overperforming to substantially underperforming when comparing it to variance reduction. Due to the lack of consistency, the hyperparameter tests of tree depth and minimum samples split do not provide trustworthy information regarding the performance of loss functions themselves. Despite that, it does not take away from the fact that the custom loss function has a potential to outperform variance reduction in certain scenarios, but merely that it is not consistent using current RF parameters. Variance reduction, on the other hand, did not have significant deviations in its performance. Its worst performing model had the sensitivity of 3.74 and best performing model had the sensitivity of 3.94.

8.2.3 Potential Solutions

In order to tackle these issues, several options may be considered. The first one being to increase the number of trees within a RF. This option is an ideal one theoretically, yet a challenging one practically. Using current custom implementation of random forest and decision trees, RF models having as low as 15 trees can take several hours to test, depending on the tree depth used. Increasing this amount will also increase the runtime, which makes the overall testing slow and impractical. A solution to it would be to optimize the code's performance. However, the current python code is already heavily optimized and uses GPU for calculating the loss function and other available helper functions. For further optimizations, the code would have to be rewritten in a more efficient language, such as C++.

Another possible solution is to reduce the training data for individual trees, which would allow using a larger ensemble of trees. This method has a risk of individual trees underfitting the data, however, it has a high potential to improve the performance at a low implementational cost as compared to rewriting the code in a different language.

A different approach to this problem is to remove the least significant features. This method is also known as *dimensionality reduction*. It is possible that, by choosing a set amount of the most important features that individual decision trees tend to pick first, the resulting RF will end up with fever individual trees that perform poorly. However, this method comes with a potential risk of a decrease in model's performance.

8.2.4 Preliminary dimensionality reduction

The Table 10 and Figure 39 below show the performance of the same five RF models, using the same hyperparameters. However, in this case, the number of available features has been reduced from 30 to 12 most commonly used ones.

	Variance Reduction	**Custom *Sum* Loss Function**
Run 1	2.63	2.92
Run 2	3.01	2.72
Run 3	2.82	2.69
Run 4	2.73	2.67
Run 5	2.83	2.84

Table 10: Deviations of peak sensitivities for RF models using 12 most important features

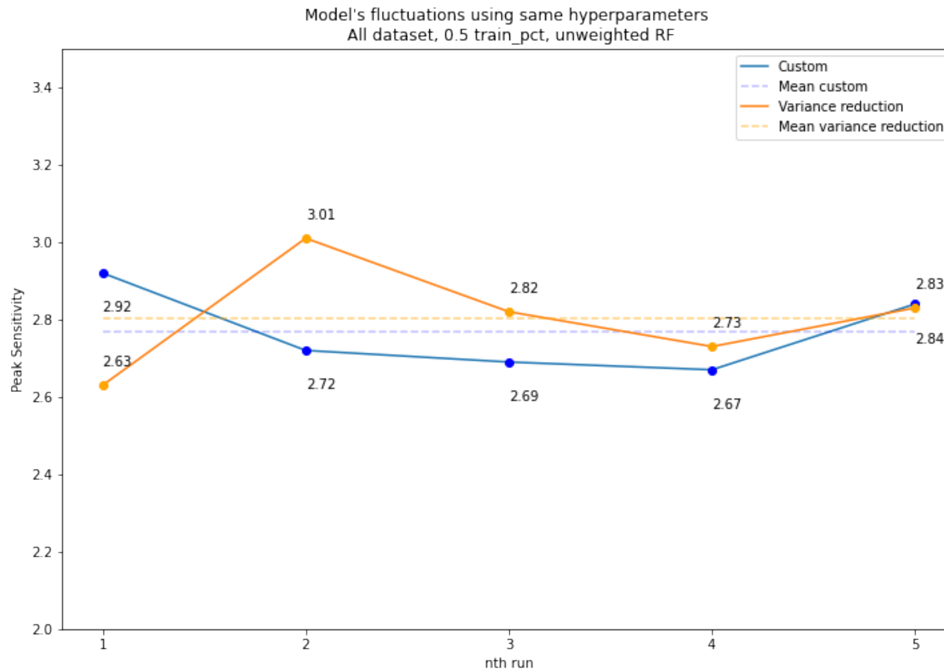


Figure 39: Deviations of peak sensitivities for RF models using 12 most important features

The initial test of reducing the amount of available features did indeed reduce the fluctuations in the peak sensitivity significantly. The difference between the maximum and minimum maximum sensitivities between the different models was merely 0.25, as opposed to 1.46 in the previous runs. However, a severe drop in the peak sensitivity has been noticed. Using a higher amount of features, the best performing model had the peak sensitivity of **4.71**, whereas using reduced features, it has dropped to **2.92**. The overall average sensitivity has dropped from **4.07** to **2.75** for the custom *sum* loss function and from **3.82** to **2.80** for the variance reduction. Such a large decrease is unlikely to be caused merely by randomness, as it appears prominent in both: variance reduction and custom loss function.

Nonetheless, the features removed were selected by manual inspection of decision trees and their features chosen within the forest, which is not optimal. Further, in-depth dimensionality reduction has been performed by first implementing feature importance calculations into the decision tree and random forest models.

8.2.5 Increasing the number of estimators to 30

The next attempted solution for increased model stability was increasing the number of estimators of a RF model by the factor of 2, while at the same time reducing the data per tree by the same factor, for the sake of runtime. It must be noted that, since the fluctuations in the variance reduction models have not been significant so far, the efficiency of increased number of estimators within a RF was only tested on the custom loss function.

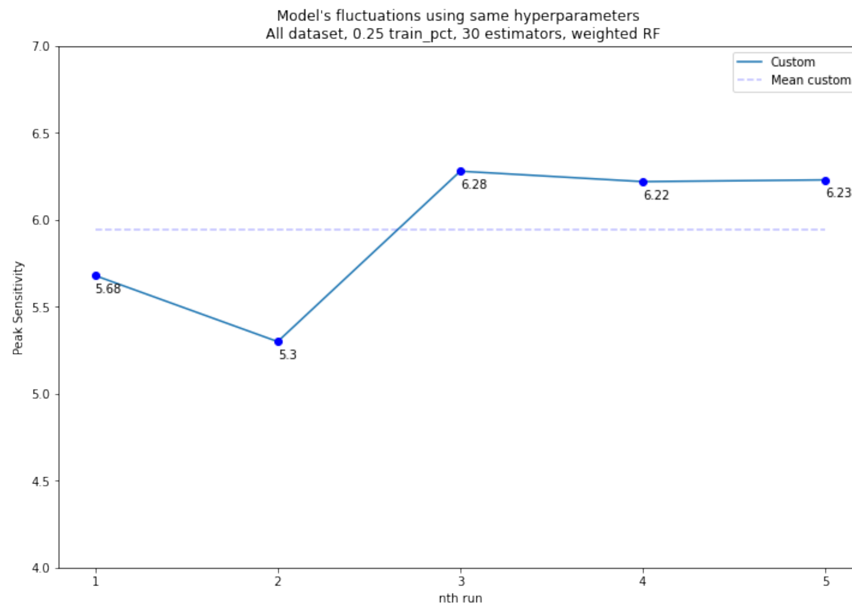


Figure 40: Deviations of peak sensitivities for RF models using 30 estimators

The Figure 40 above displays fluctuations of models using custom *sum* loss function. The number of estimators has been increased to 30 from 15, and the training data percentage has been reduced to 25% from 50%. Despite the higher amount of estimators, the models still had relatively high fluctuations in the peak sensitivities achieved. Nonetheless, a noticeable improvement in the sensitivities produced by the models using this combination of RF hyperparameters can be observed.

8.2.6 Balanced dataset and alternative weighting

As the results did not yet indicate consistency between RF models with same parameters, further testing and other improvements had to be considered. One of them was to use a balanced dataset. Since RF models use a random sampling of the data, selected from the entire train dataset, it was expected for models to have different data distributions. This is not an issue inherently. In fact, it is

one of the advantages of the random forest algorithm, as it provides additional diversity between the trees. Nonetheless, an important fact to consider is that, while the goal of the model is to differentiate between the background and the signal data, the background data is sampled from a set of individual background files, corresponding to different simulated particle decays. Hence, choosing data completely at random has a potential to cause a discrepancy between RF models' performance, especially if a relatively low amount of estimators is used.

In order to test the impact this idea has on the RF models' performance and their fluctuations, a different approach of data sampling has been tested. Instead of selecting data completely at random, it was selected similarly as when splitting the data into training, validation and test datasets. Each of the background data sample was given a unique identifier, which allowed to select the same percentage of data from each of individual samples. As a result, each tree within a RF model will have an equal amount of each of the background data type, while still choosing random samples. Additionally, it also ensured that each tree had a some signal data in it.

Moreover, a different approach to tree voting has been tested. Instead of each tree's vote being equal, their votes were weighted based on their peak sensitivities. This means that, when making the final decision whether a sample is signal or background, the votes of the trees that performed well will be more impactful in comparison to trees that performed poorly.

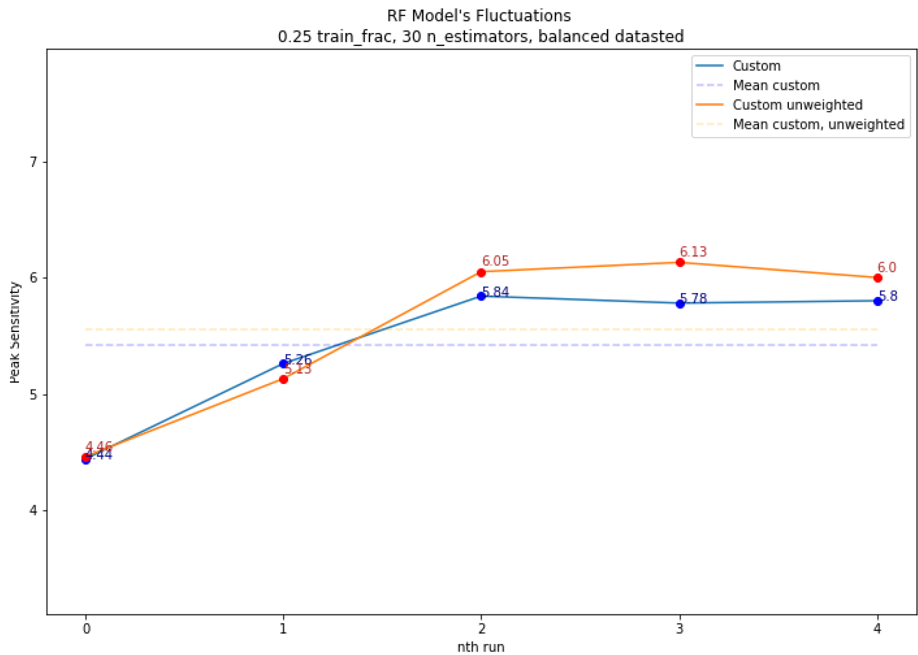


Figure 41: Deviations of peak sensitivities for RF models using balanced dataset sampling

The Figure 41 above displays the deviations of RF models using the same depth and minimum split parameters in addition to 30 estimators. The models were trained with a balanced dataset consisting of 25% of the total training data. The blue plot represents the models trained with their votes weighted according to their sensitivity, and the yellow plot represents the models with regular voting. The results were unanticipated, as the difference in the peak sensitivities between the best and worst performing models increased in comparison to unbalanced datasets. The difference between the lowest and highest performing models' sensitivities was as large as *1.6*, whereas using the unbalanced dataset the difference was *0.9*. There is a possibility of the lowest performing model being a heavy outlier, as no other models using the same parameters were observed to have such a low peak sensitivity. However, even if this was the case, the peak sensitivities would still be approximately that of unweighted dataset in this case. Hence, using this particular combination of hyperparameters, no observations of improvement of balanced dataset were noticed. There were also no noticeable improvements in different tree vote weighing during this test.

8.2.7 Feature importance

It is possible to get a better insight into each individual RF model's decision-making process by the use of feature importance. The feature importance was calculated by analysing the features chosen of each individual tree within the forest. When a split was made within a tree, the selected feature was assigned a value, equal to the loss function's calculation. For custom loss functions, this meant sensitivity increase, and for variance reduction it meant the decrease in variance between children and parent nodes. This means that, a feature with a high feature importance value will be one that was chosen often and provided a high increase in sensitivity, whereas a feature with a low feature importance value was either not chosen often or did not provide a significant increase in sensitivity.

Figure 42 displays an example of feature importance graph, showing feature importance from the first model of Figure 41.

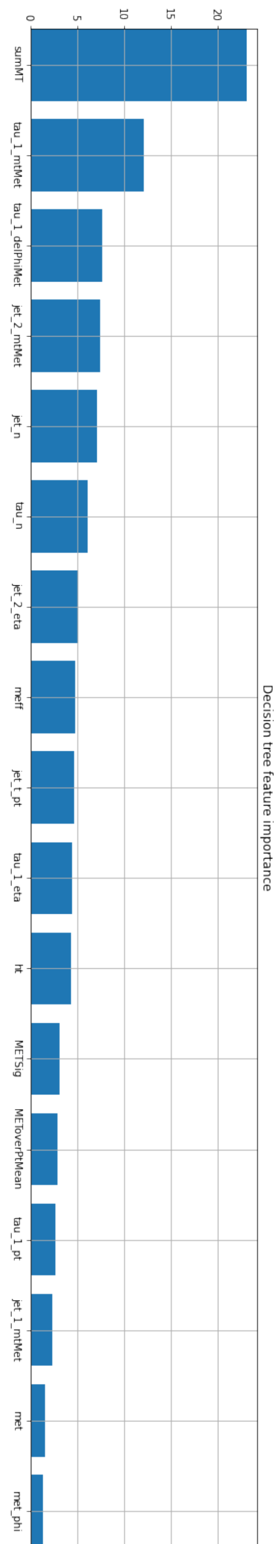


Figure 42: An example of initial feature importance plots

8.2.8 Increasing the number of estimators to 60

The fluctuations with balanced and unbalanced datasets have also been tested using 60 estimators, training each estimator with 12.5% of training data. The results of this test are summarized in Figure 43.

This combination of tested parameters provided most consistent results, while not having a negative impact regarding the peak sensitivity achieved. Using a balanced dataset, represented by the blue plot in the figure, the difference in peak sensitivity achieved between the worst and best performing models was equal to 0.79. Using an unbalanced dataset, represented by the orange plot in the figure, the difference was even lower at 0.55.

Using a balanced dataset did not seem to provide noticeable, if any, improvements in achieving more consistent results. It also provided worse average results, however, the difference was not significant. Nonetheless, RF models with a balanced dataset had the mean sensitivity of 6.15, whereas the models with an unbalanced dataset had an average of 6.33. The difference in sensitivities could very well be caused by the fluctuations of the models, however, since using a balanced dataset did not seem to provide any improvements, an unbalanced dataset will be used for upcomming tests.

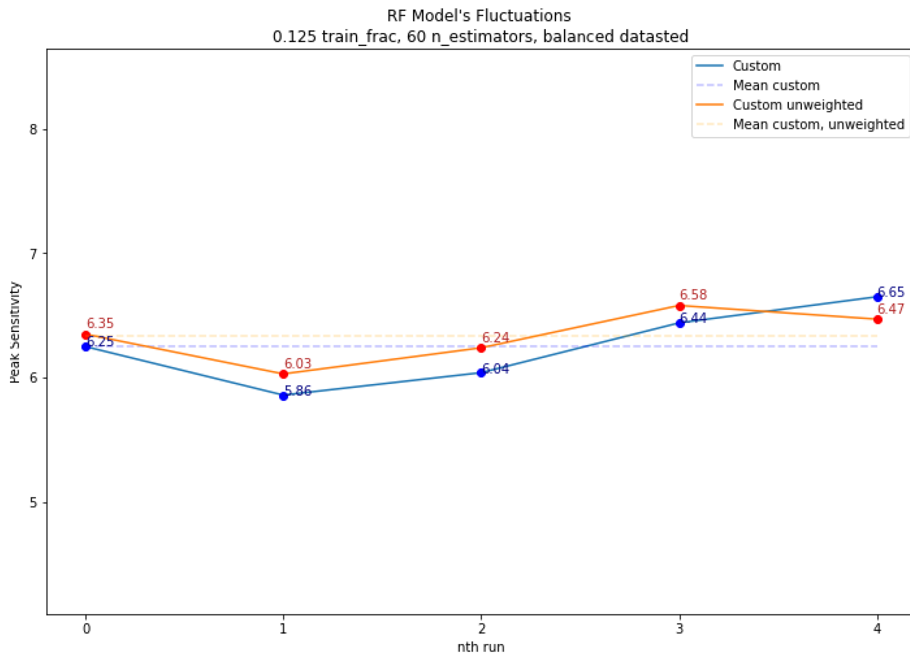


Figure 43: Deviations of peak sensitivities for RF models using 60 estimators

Another point to notice is that, while the sensitivities had relatively low fluctuations, the feature importance graphs for the best and worst performing models were quite different. In the worst performing model, the feature *tau_1_mtMet* was the most important one, closely followed by *sumMt*. In the best performing model, *sumMt* feature had a clear difference in its importance among other features. Otherwise, with some exceptions, the selection of the most important features was similar among the models.

The fluctuations of non-sum version of custom loss function have also been tested. However, as it did not provide adequate results when being used as a classification model, it has been used as a regression model using different thresholds for classification instead. The model has been tested with 30 estimators and 25% of data per model, and 60 estimators and 12.5% of data per model.

As seen in Figure 44 and Figure 45, the fluctuations were not as severe as when using *sum* custom loss function in both cases. The difference in peak sensitivities using 30 estimators was merely *0.22*, however it increased to *0.775* using 60 estimators. Based on this information, it can be concluded that 30 estimators using 25% provides adequate and consistent results using the non-sum version.

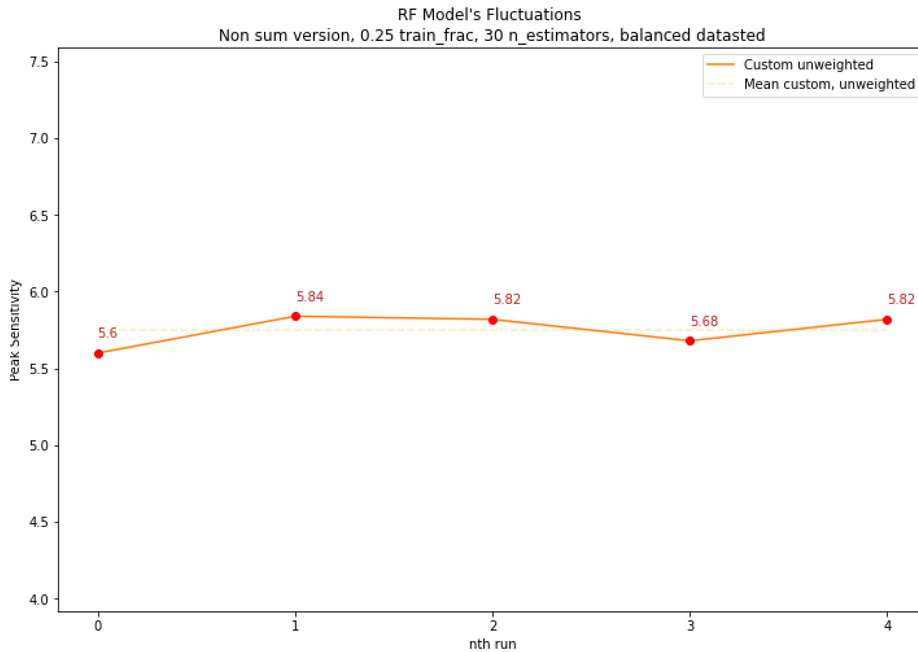


Figure 44: Sensitivity deviations using *custom non-sum* loss funcion with 30 estimators

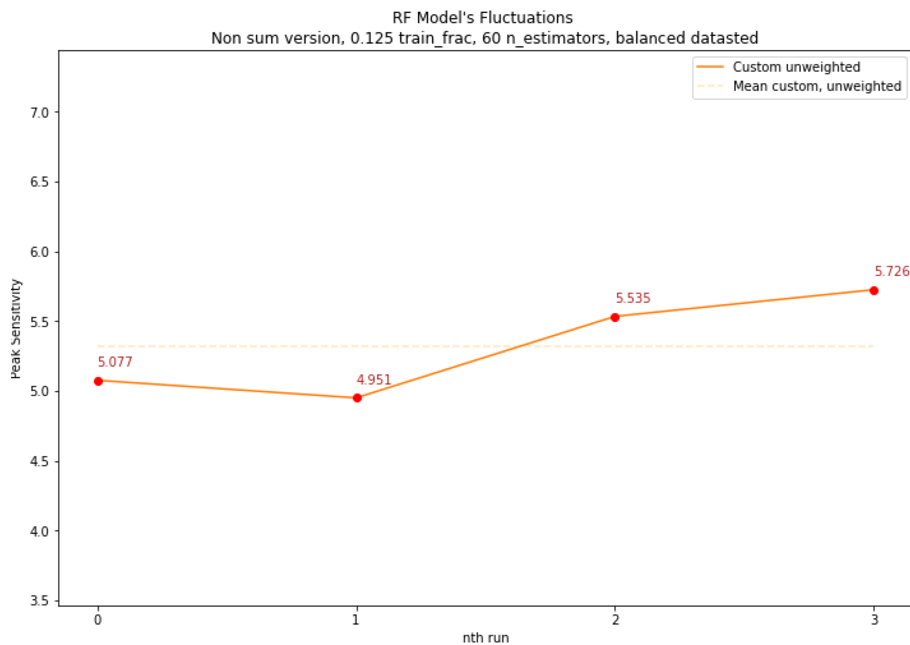


Figure 45: Sensitivity deviations using *custom non-sum* loss funcion with 60 estimators

8.2.9 Remarks

While the tests described above did provide an insight into RF models' consistency, several points have to be taken into an account. The computational limitations were always a key factor in deciding models' parameters. Each of the fluctuation test has been performed by the use of same decision tree parameters: *tree depth* of 3 and *minimum samples to split* of 20.

The reason behind choosing this particular set of decision tree parameters is that, due to the previous tests performed, it was known that RF models with these parameters were able to provide adequate results for all the loss functions tested. In addition, the tree depth has a major impact on the runtime of models. With the current depth and other RF hyperparameters as described above, a five model fluctuation test has a runtime of approximately 5 hours. That means, testing the fluctuations for 2 different loss functions has the runtime of approximately 10 hours for a single combination of RF hyperparameters.

Had the runtime not been a limiting factor, more adequate tests could be performed. This includes testing different depths and minimum splits, different training percentages for trees, different amount of features chosen and additional runs, which would provide definitive results. However, this was not feasible due to runtime constraints. Conversely, if the runtime was not an issue, there would also be no need to test the consistency of the models. Random forest is inherently supposed to fluctuate to a certain extent. By minimizing those fluctua-

tions, it is possible that the peak performance could also be potentially reduced. At the end of the day, it is desired to find a model with the best performance, despite the fluctuations. The model's state, its chosen data and features for each tree could then be easily extracted, replicating the best performing model, which could then be used on real data.

In a theoretical scenario where runtime was not an issue, the preferred approach in tackling the inherent inconsistencies of RF models would be to create multiple models for each set of hyperparameters tested and choose the best performing one. However, this is not feasible with the current constrains.

It is also important to note that RF model performance depends not only on its own parameters, but also greatly on the parameters of the individual decision trees it consists of. The following parameters are the main ones contributing to the model's performance:

- The number of estimators of in a random forest.
- The train percentage of individual trees within the forest.
- The maximum depth of the trees within the forest.
- The minimum amount of samples split for trees within the forest.

If we were to test 3 different parameters for each of the point listed above, that would make 3^4 or 81 combinations. Testing 5 models for each of them and picking the best performing one, would be equivalent to making 405 models. Additionally, the tests would have to be performed using 3 different loss functions, resulting into 1215 total models. If the approximate runtime of a single model, which also depends on the parameters themselves, is 40 to 60 minutes, testing would require 48 600 minutes or approximately 34 days.

Fluctuation testing provided a certain level of guarantee of the model's performance, taking out two of the parameters out of the equation. Nonetheless, this does not mean that the fluctuations are non-existent. Rather, they are anticipated to be low enough to the point where conclusions can be made regarding the loss function themselves and not the parameters of the RF.

8.3 Hyperparameter testing using reduced fluctuations

Using information gained from the fluctuation testing, it was now possible to test other hyperparameters, namely tree depth and minimum samples split, with a higher level of confidence for the result consistency. Parameters of number of estimators and train percentages were chosen accordingly to ones providing the least fluctuations and best performance for each of the loss function. In total, the following loss functions and RF parameters were tested:

- Custom sum loss function, unweighted tree voting
- Custom sum loss function, weighted tree voting
- Custom non-sum loss function
- Variance reduction using 30 estimators and 25% train percentage
- Variance reduction using 60 estimators and 12.5% train percentage

For each of the loss function, a combination of depths of 3, 5 and 7, and minimum samples split of 2, 10, 50 and 100 were tested, making a total of 12 models per loss function.

8.3.1 Results

The tables below summarize the peak sensitivities achieved with a specified combination of depth and minimum samples split parameters.

depth\min_split	**2**	**10**	**50**	**100**
3	6.23	**6.60**	5.85	6.13
5	6.06	6.10	6.10	5.89
7	6.10	6.30	6.40	6.15

Table 11: Hyperparameter test results using weighted custom *sum* loss function

depth\min_split	**2**	**10**	**50**	**100**
3	6.37	**6.69**	6.00	6.08
5	6.24	6.16	6.24	6.04
7	6.20	6.34	6.38	6.25

Table 12: Hyperparameter test results using unweighted custom *sum* loss function

depth\min_split	**2**	**10**	**50**	**100**
3	5.75	5.86	5.80	5.39
5	5.67	5.98	**6.16**	5.41
7	5.18	5.96	6.09	**6.16**

Table 13: Hyperparameter test results using custom *non-sum* loss function

depth\min_split	**2**	**10**	**50**	**100**
3	5.20	5.55	5.11	5.01
5	4.99	**5.95**	4.54	5.91
7	5.18	4.76	5.36	5.00

Table 14: Hyperparameter test results using variance reduction with 30 estimators and 25% train data per tree

depth\min_split	**2**	**10**	**50**	**100**
3	4.72	5.59	5.62	5.70
5	4.96	5.24	5.57	**6.08**
7	4.00	4.30	5.58	5.95

Table 15: Hyperparameter test results using variance reduction with 60 estimators and 12.5% train data per tree

Loss function	**Parameters**	**Sensitivity**	**s_s**	**b_s**
Weighted custom *sum*	depth: 3, min_split: 10	6.60	120	203
Unweighted custom *sum*	depth: 3, min_split: 10	**6.69**	126	228
Custom *non-sum*	depth: 5, min_split: 50	6.16	154	470
Weighted variance reduction	depth: 5, min_split: 10	6.08	140	390

Table 16: RF model performance comparison using different loss functions and corresponding best *depth* and *minimum samples split* hyperparameters

Table 16 summarizes the results from all the tables shown above. The best performing model using custom loss function had peak sensitivity equal to **6.69**, whereas the best performing model using variance reduction had the peak sensitivity of **6.08**. This means that the custom loss function did provide an increase in the peak sensitivity achieved by approximately **10.03%**. Additionally, the custom loss function had a severe decrease in background incorrectly classified as signal. The amount has decreased by approximately **58.46%**, whereas the amount of signal correctly classified as signal has only decreased by **10.0%**.

Another point to notice is the average sensitivity for any combinations of parameters. Using the custom *sum* loss function, only two of the models with different combinations of hyperparameters achieved sensitivity less than 6.00, as seen in Table 12. In comparison, out of all the models using variance reduction, only a single one was able to achieve sensitivity higher than 6.00, as seen in Table 15 and Table 14.

To ensure that the difference in sensitivities for different loss functions is not caused by fluctuations, the models using parameters of best performing models were trained and tested multiple times, similarly as in fluctuation testing.

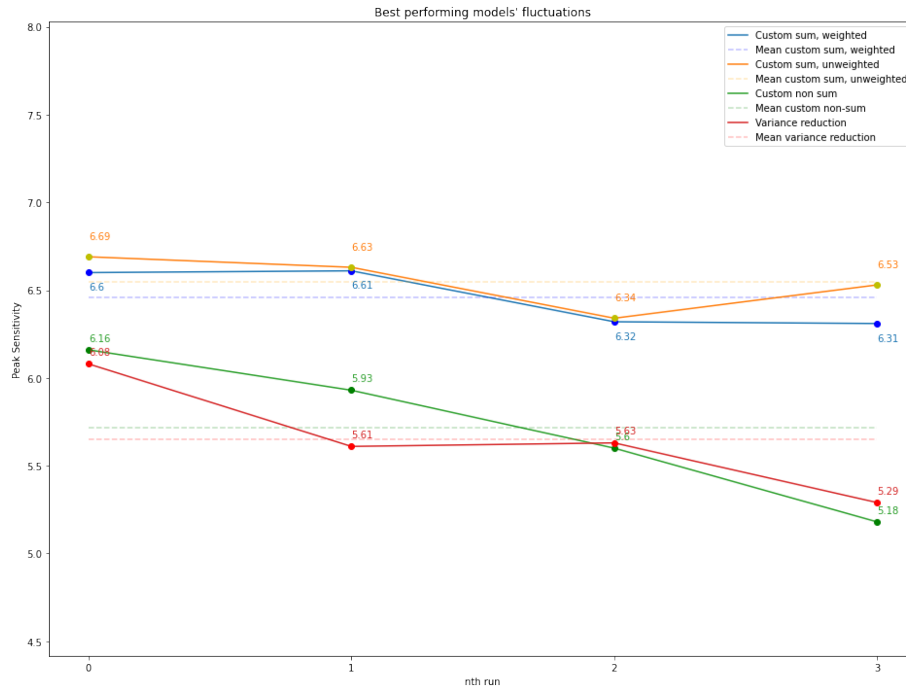


Figure 46: RF model deviations in peak sensitivities using different loss functions and corresponding best hyperparameter combinations

As seen from the Figure 46 above, noticeable fluctuations in the performance of models of non-sum custom loss function and variance reduction can be observed, despite the models having the least fluctuations during the fluctuation testing. Including the first tests, the peak sensitivity of best-performing variance reduction model ranged from 5.29 to 6.08 and the sensitivity of best performing model using custom loss function ranged from 6.34 to 6.69. The most likely explanation for it is the fact that the fluctuation testing was performed with a set depth and minimum samples split parameters in order to keep a reasonable experimentation runtime. Therefore, by changing these parameters in hyperparameter testing, additional fluctuations may occur.

Nonetheless, with the extra confidence provided in the results of different loss functions, it is now possible to compare models' feature importance distributions. There exists a noticeable difference in the features selected by the model using variance reduction, however, the difference is not as prominent between the different versions of custom loss function.

Additionally, the feature values in feature importance plots are now averaged according to the number of times a feature has been randomly selected. This provides more consistent results, by separating feature importance from the randomness of the random forest.

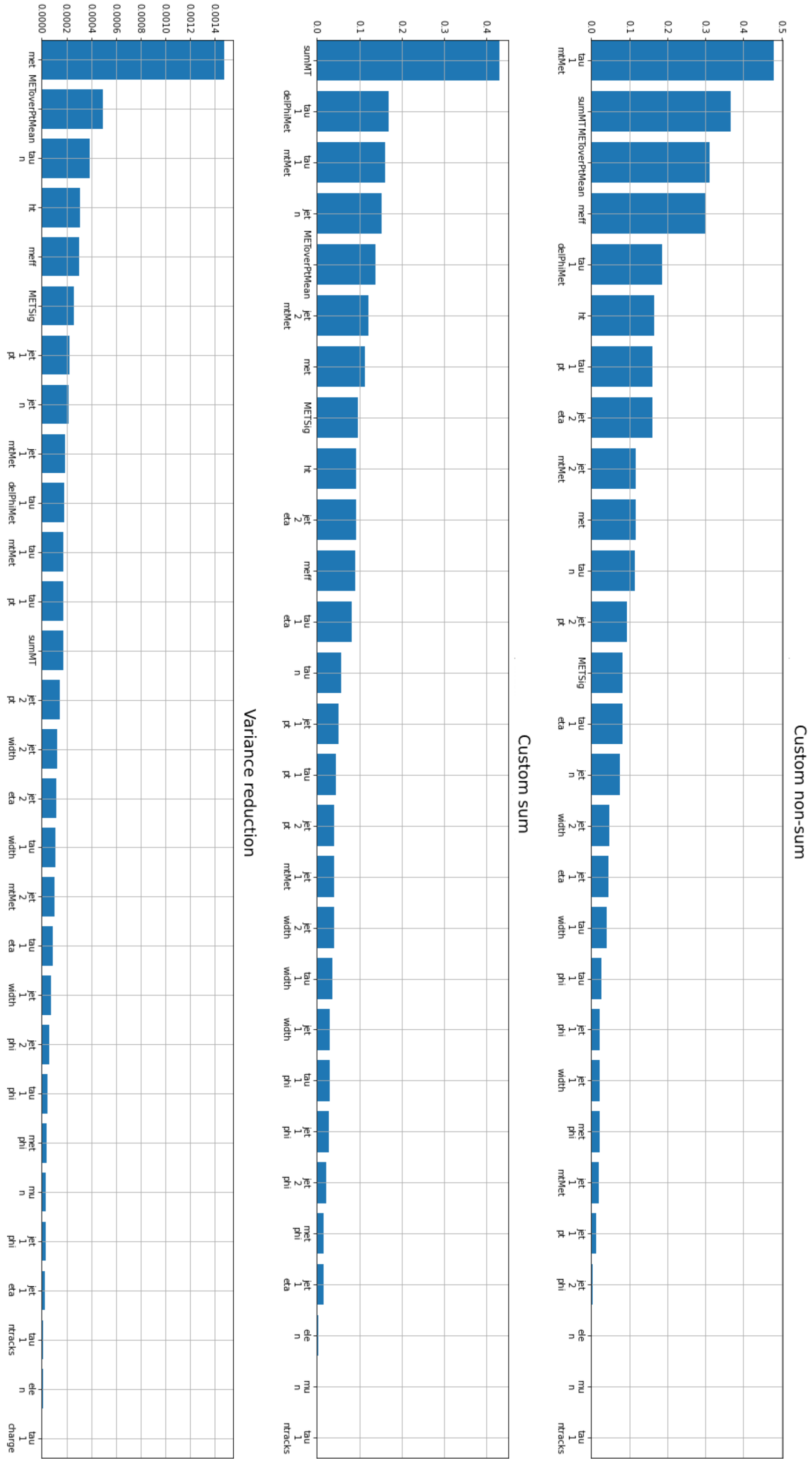


Figure 47: Feature importance distributions for different loss functions

8.4 Increasing the number of features

Previous random forest tests have been performed using a set amount of features equal to $\lceil\sqrt{p}\rceil$, where p is the number of total available features. Since 30 initial features were used, this resulted into each tree being trained with 6 features each. However, the number of features per tree is also a parameter to be tested, as it can directly influence the model's performance.

The effect increased number of features per tree has on the final performance of the model has been tested by increasing the number of features to 9 and 15. The test involving 9 features per tree has been performed with maximum depths of 3, 5 and 7, and the best matching minimum samples split parameter, based on the performance in the prior hyperparameter testing. For models using 15 features per tree, the test was limited to a single combination of depth and minimum samples split, due to runtime constraints. The results of this test are summarized in the tables below.

parameters\number of features	**6**	**9**	**15**
depth: 7, min_split: 100	6.16	6.07	5.95
depth: 5, min_split: 50	6.16	5.47	x
depth: 3, min_split: 10	5.86	5.26	x

Table 17: RF model performance comparison with different amount of features selected per tree, using *custom non-sum* custom loss function.

parameters\number of features	**6**	**9**	**15**
depth: 7, min_split: 50	6.38	6.27	5.75
depth: 5, min_split: 2	6.24	4.92	x
depth: 3, min_split: 10	6.69	6.03	x

Table 18: RF model performance comparison with different amount of features selected per tree, using *custom sum* loss function

parameters\number of features	**6**	**9**	**15**
depth: 7, min_split: 50	6.40	6.31	6.37
depth: 5, min_split: 2	6.06	4.69	x
depth: 3, min_split: 10	5.60	5.86	x

Table 19: RF model performance comparison with different amount of features selected per tree, using *custom sum* custom loss function with weighted voting.

parameters\number of features	**6**	**9**	**15**
depth: 7, min_split: 100	5.95	5.80	5.74
depth: 5, min_split: 10	5.57	4.73	x
depth: 3, min_split: 50	5.59	5.84	x

Table 20: RF model performance comparison with different amount of features selected per tree, using variance reduction

Before assessing the results, it is important to note that, if sensitivity differences between models trained with different amount of features per tree are not substantial enough, they can be outweighed by the model's fluctuations. To minimize this risk, the test has been performed for several sets of hyperparameters, which reduces the effect of model fluctuations. Nonetheless, there was no noticeable improvement in models' performance by increasing the amount of features per tree.

8.4.1 Dimensionality reduction utilizing feature importance

Dimensionality reduction has been attempted before. However, there are a couple of issues with the initial approach. Most importantly, the custom implementation lacked the option of calculating the feature importance. Without accurate feature importance information, it cannot be guaranteed that the removed features were, in fact, insignificant. Moreover, the amount of features removed was quite drastic, despite them seemingly being not important by manual inspection. Therefore, a different approach has been taken this time. Instead of selecting a number of best performing features, a number of the worst features has been removed, in accordance to the feature importance information acquired from the previous analysis. By utilizing feature importance plots, the likelihood of removed features being important was substantially reduced.

Using this approach, 6 features were removed, all of which had a low importance value for every of the loss function used. The removed features were the following: *tau_1_ntracks, mu_n, ele_n, jet_1_eta, met_phi, tau_1_phi, tau_1_charge*.

Using the reduced features, model tests have been performed using the best-performing combination of hyperparameters for each of the loss function used. The results are summarized in Table 21.

	Reduced features	**All features**
Custom non-sum	5.82	6.16
Custom sum, unweighted	6.35	6.69
Custom sum, weighted	6.17	6.60
Variance reduction	4.89	6.08

Table 21: RF performance comparison using reduced features (-6)

The resulting peak sensitivities using reduced features were slightly lesser in comparison to using a full feature set. However, an important factor has to be taken into a consideration. As observed from the previous fluctuation testing, the sensitivities tend to fluctuate even with same parameters, hence fluctuation testing was also performed with the reduced features, as seen in Figure 48.

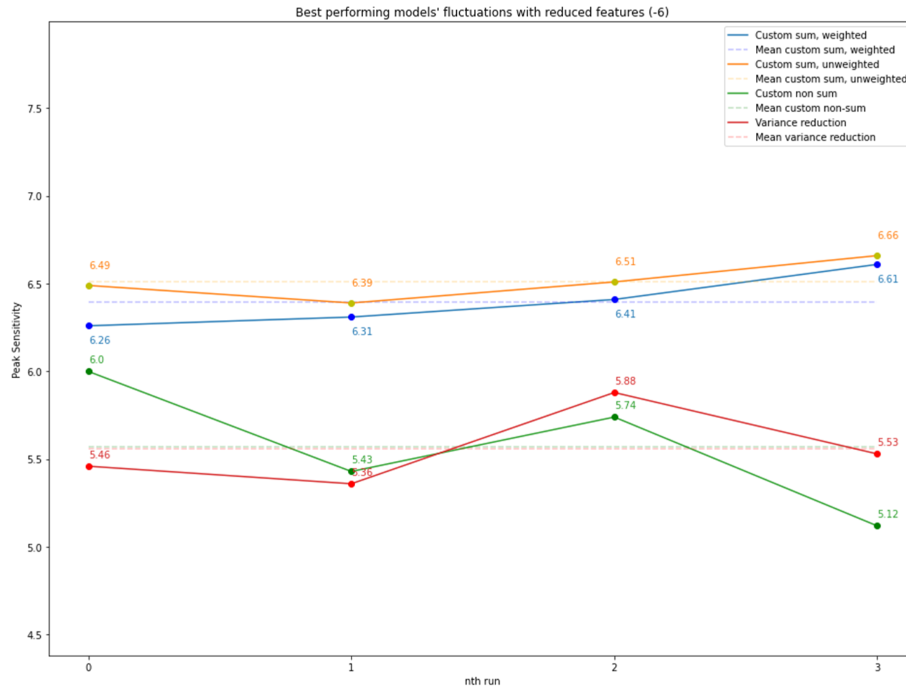


Figure 48: Deviations of peak sensitivities for RF models with reduced features (-6)

Comparing Figure 48 and Figure 46, it can be noticed that the decrease in the peak sensitivities for different loss functions has not been significant. For custom *sum* loss function, the decrease has been as low as 0.03, which is completely negligible. The decrease in the peak sensitivity for custom *non-sum* loss function was slightly higher, at 0.16, and the decrease for variance reduction was the highest, at 0.2. Nonetheless, the decreases are more or less negligible and could entirely be caused by model fluctuations.

The fluctuations, i.e. the difference in the sensitivities of worst and best performing models, changed from 0.35 to 0.27 for the sum custom loss function, from 0.98 to 0.88 for the non-sum custom loss function, and lastly from 0.77 to 0.52. Hence, the overall performance has not been affected greatly, while there was a noteicable decrease in fluctuations.

Further dimensionality reduction was performed by removing 12 of the least significant features. However, it resulted into an approximate 7 to 10 percent decrease in peak sensitivities achieved, depending on the loss function used, without noticeable fluctuation decrease as compared to the removal of 6 most insignificant features.

8.5 Feature importance validation

Feature importance places a crucial role for dimensionality reduction, which ultimately provides more consistent models and thereby more consistent results. Even if the initial ranking of features is consistent with expectations, additional tests can be performed to verify the feature importance itself.

Differentiating the different background and signal files analytically, based on their features and distributions, is a challenging task, which is why machine learning is needed in the first place. However, this difference can be much more prominent when comparing different background types and their feature distributions. By training a ML model to differentiate between two different background types, rather than a signal and a combination of all the background types, the difference in the resulting feature distributions can be observed analytically. In this case, the distributions of most important features, as according to their feature importance plots, are expected to be noticeably different between the different background types. Likewise, features with low feature importance values are not expected to have a noticeable difference between the two different background files.

In order to perform this test, two background types have been selected - *ttbar* and *wtaunu*. Several models have then been developed and trained using variance reduction, custom *sum*, and custom *non-sum* loss functions. For each of them, the following hyperparameters have been tested:

- Minimum samples split: 10, maximum depth: 3
- Minimum samples split: 50, maximum depth: 5
- Minimum samples split: 100, maximum depth: 7
- Minimum samples split: 100, maximum depth: 9

Additionally, each RF model chose 6 features per tree, had the number of estimators equal to either 60 or 30, and data percentage per tree equal to either 12.5% or 25%, depending on the parameters that maximize the loss function used.

Using multiple models provided additional confidence regarding the consistency of features chosen, as it can fluctuate due to randomly selected features. Figure 49 below shows typical distribution of feature importance for each of the loss functions.

Figure 49: Feature importance distributions for different loss functions using *ttbar* and *wtaunu* data files

As observed from the Figure 49, there are noticeable differences between feature importance distributions for different loss functions. This is expected to an extent, as the loss functions maximize different properties. Another noticeable difference is the amount of features in feature importance graphs for models using different loss functions. By using only two different background types and differentiating between them, the custom loss functions were not able to find improvements with several of the available features, hence they are not displayed in the feature importance plots. The *sum* custom loss function was able to find improvements with 20 of 31 possible features, non-sum custom loss function 15, and the variance reduction was able to attain some improvements with all the available features. In spite of that, the improvements for a significant portion of available features seemed to be negligible.

Figure 56 shows feature distribution plots for features that, in general, had an overall large feature importance values over all the loss functions tested. Data from *ttbar* background is displayed as a blue, filled histogram, and the data for *wtaunu* is displayed as a yellow step histogram. The plots are weighted in accordance to simulation weights in addition to being normalized, to account for different sample sizes.

Nonetheless, if it is possible to differentiate between the two background types analytically, there should be a region in the graphs, where there is little to no overlap between them. This observation is somewhat prominent in several of the feature plots displayed in Figure 56, where the right-most regions tend to contain some of *wtaunu* samples, and low amounts of *ttbar* samples. However, it must also be taken into a consideration that the y-scale of the plots is logarithmic. Since, in this case, distributions of continuous features tend to fall down as they approach their maximum values, and the fact that this is also the region where the difference between two background distributions is most prominent, the difference may not be as large as it may appear initially.

By looking at Figure 63, which displays feature distributions over the lowest scoring features, an immediate difference can be observed from the features displayed in the Figure 56. Distinguishing the two background files based on these features analytically becomes vastly more difficult. For the majority of the plots, *ttbar* almost completely overlaps with *wtaunu*, making their distinction next to impossible. Another observation to be made is that many of the poorly performing features were discrete, unlike the features that performed well. Nonetheless, these features were selected infrequently by the decision trees and if they were, they did not provide significant improvements.

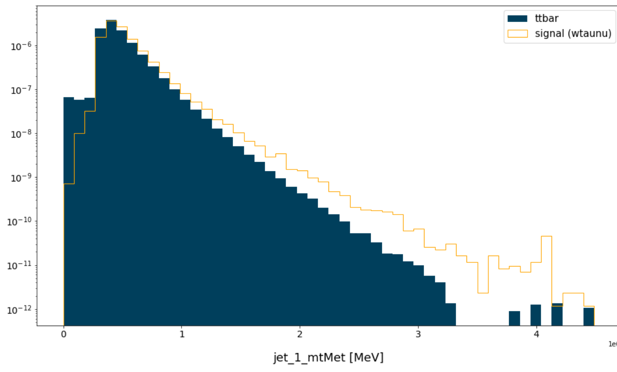


Figure 50: Feature importance validation *jet_1_mtMet*

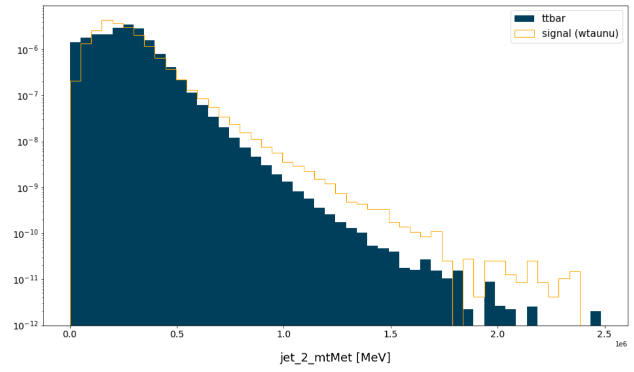


Figure 51: Feature importance validation *jet_2_mtMet*

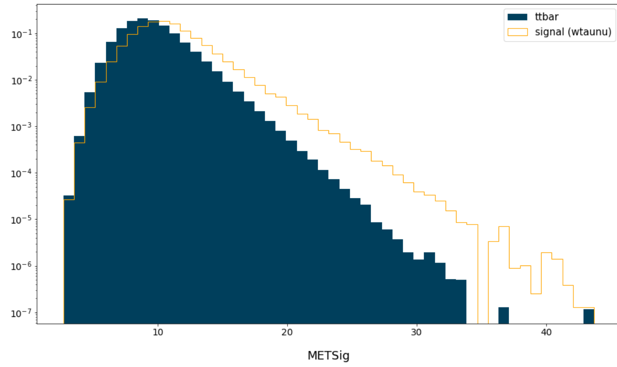


Figure 52: Feature importance validation *METSig*

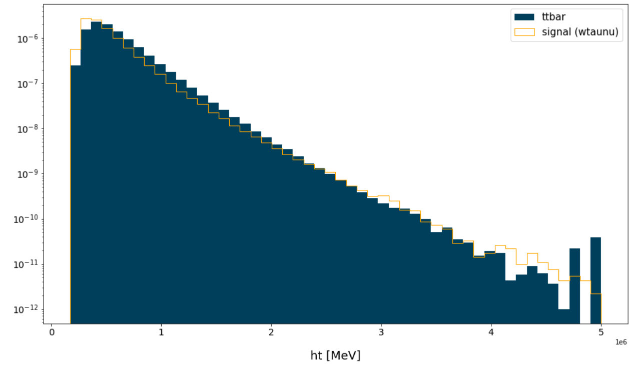


Figure 53: Feature importance validation *ht*

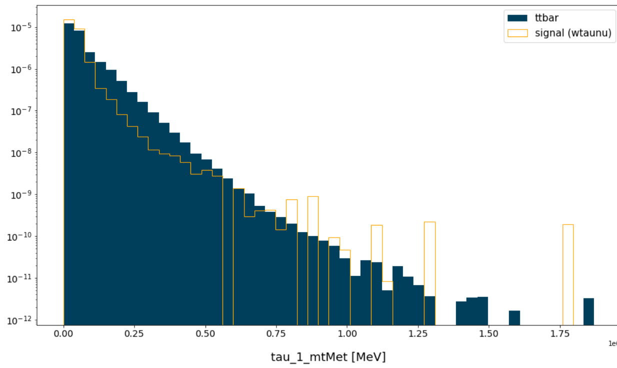


Figure 54: Feature importance validation *tau_1_mtMet*

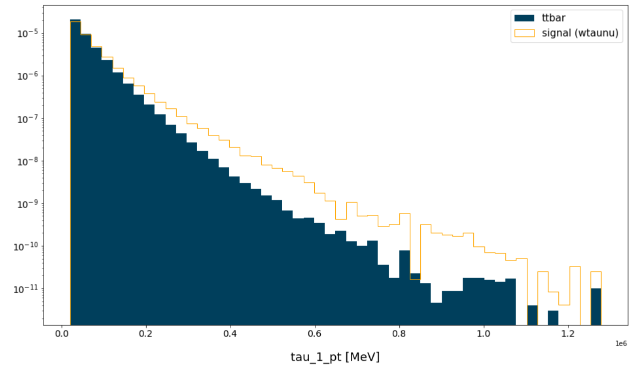


Figure 55: Feature importance validation *tau_1_pt*

Figure 56: Feature distributions of best performing features for *wtaunu* and *ttbar* datasets

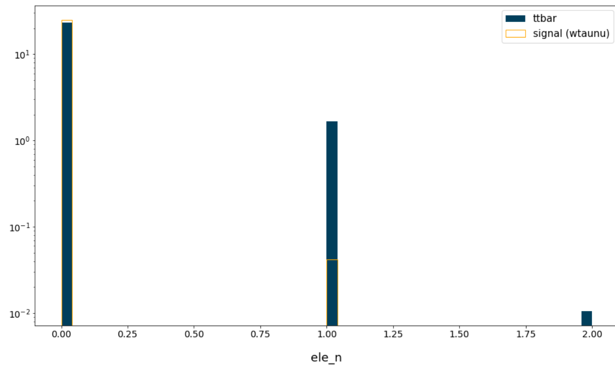


Figure 57: Feature importance validation
ele_n

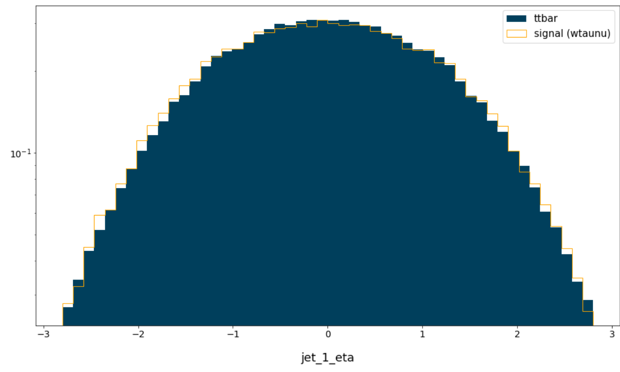


Figure 58: Feature importance validation
jet_1_n

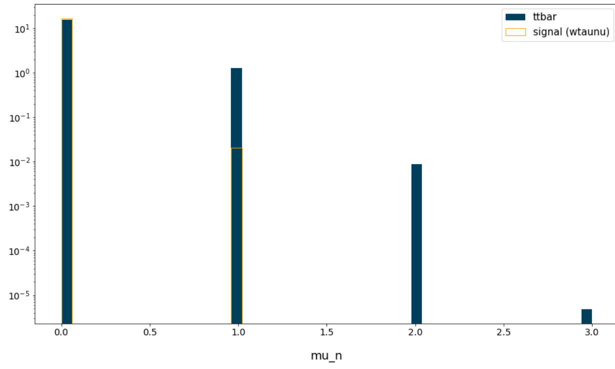


Figure 59: Feature importance validation
mu_n

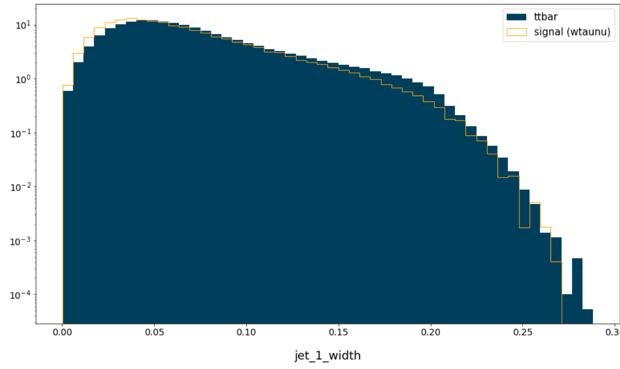


Figure 60: Feature importance validation
jet_1_width

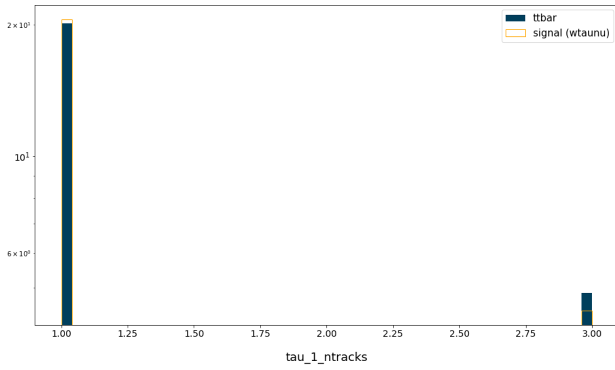


Figure 61: Feature importance validation
tau_1_ntracks

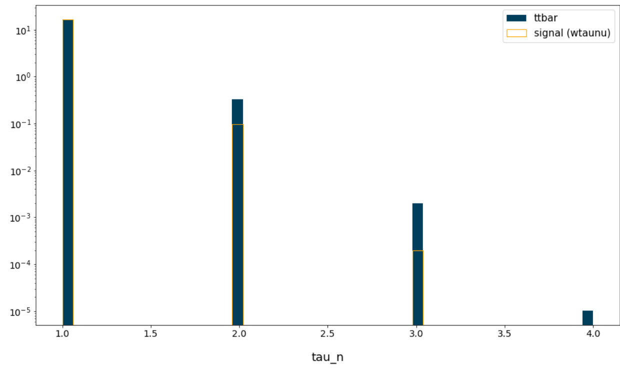


Figure 62: Feature importance validation
tau_n

Figure 63: Feature distributions of worst performing features for *wtaunu* and *ttbar* datasets

It should be noted that this test has been performed using two different background files, which is not the case in models whose intention is to distinguish between simulated signal and all background files. However, this test still increase the confidence in the feature importance rankings, and based on it, we can conclude that they appear to be reliable.

8.6 Alternative custom loss function

The details of the custom loss function have been described in the Section 6.1.2, however, in general, it maximizes the ability to separate signal from background as seen in the Equation (5). It is, however, a widely used approximation of formula known as Asimov significance [27], as described in Equation (6).

$$\frac{s}{\sqrt{s+b}} \tag{5}$$

$$\sqrt{2[(s+b)ln(1+\frac{s}{b})-s]} \quad , \tag{6}$$

where:

$s =$ Signal, correctly classified as signal

$b =$ Signal, incorrectly classified as background

Maximizing the non-approximated equation should, in theory, provide similar performance results, in addition to providing a sanity check regarding the custom loss function's performance.

To test this hypothesis, regression and classification models have been trained and tested following the same implementational principles as with previous tests, with the exception of maximizing Equation (6), rather than Equation (5). The depth, minimum sample split, number of estimators and train fraction parameters have all been chosen based on the ones maximizing the performance of the function's approximation. The parameters used for RF models are summarized in the table below.

	Regression Models	**Classification models**
Minimum sample split	50	50
Maximum depth	5	5
Number of estimators	30	60
Train percentage	25%	12.5%

Table 22: Hyperparameter summary for RF models using alternative custom loss function

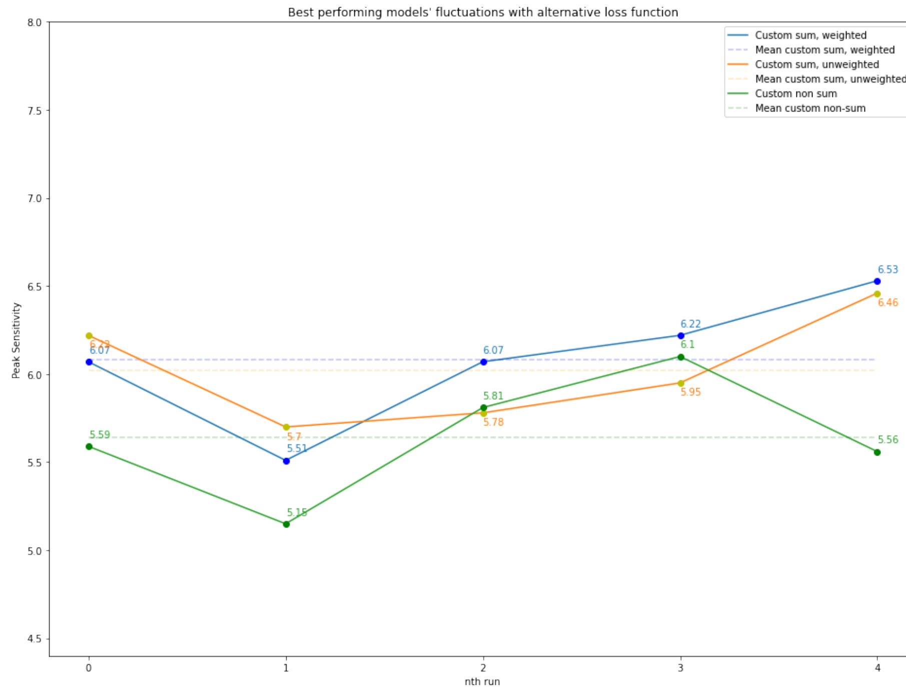


Figure 64: RF classification models' fluctuations in peak sensitivities using alternative custom loss functions, as described in Equation (6).

Since the complete loss function does not inherently differ from its approximation to the point where the difference in its results is expected to be of any significance, a very thorough hyperparameter testing was not performed. Rather, the models with the aforementioned parameters were tested multiple times, to account for the fluctuation.

In terms of the peak sensitivities achieved, both of the functions had an on-par performance. Based on previously performed tests, classification models using the approximated function were able to achieve the sensitivity of 6.69. In comparison, the classification models using complete function were able to achieve the sensitivity of 6.53. The difference is not substantial enough to the point where it could not stem from the model's fluctuations. Similarly, the difference in the peak sensitivity achieved by regression models' was as low as 0.06, with a slight advantage going to the approximated function. It is, of course, possible that the difference could be somewhat more apparent if a thorough testing would be performed, including testing different depths, minimum samples split, different amounts of estimators and more. However, significant differences are not expected, hence, also considering time consumption and the intention of this test, a thorough hyperparameter testing has not been performed with this function.

Furthermore, comparing the functions' feature importance in Figure 65, con-

siderable similarities in the feature distributions between the different loss functions can be noticed.

Considering the fact of low fluctuations in peak performance of the two loss functions, and no significant differences in feature importance graphs, as seen in Figure 64 and Figure 65, it can be concluded that the performance of both of the functions is borderline equivalent, which confirms the initial expectations.

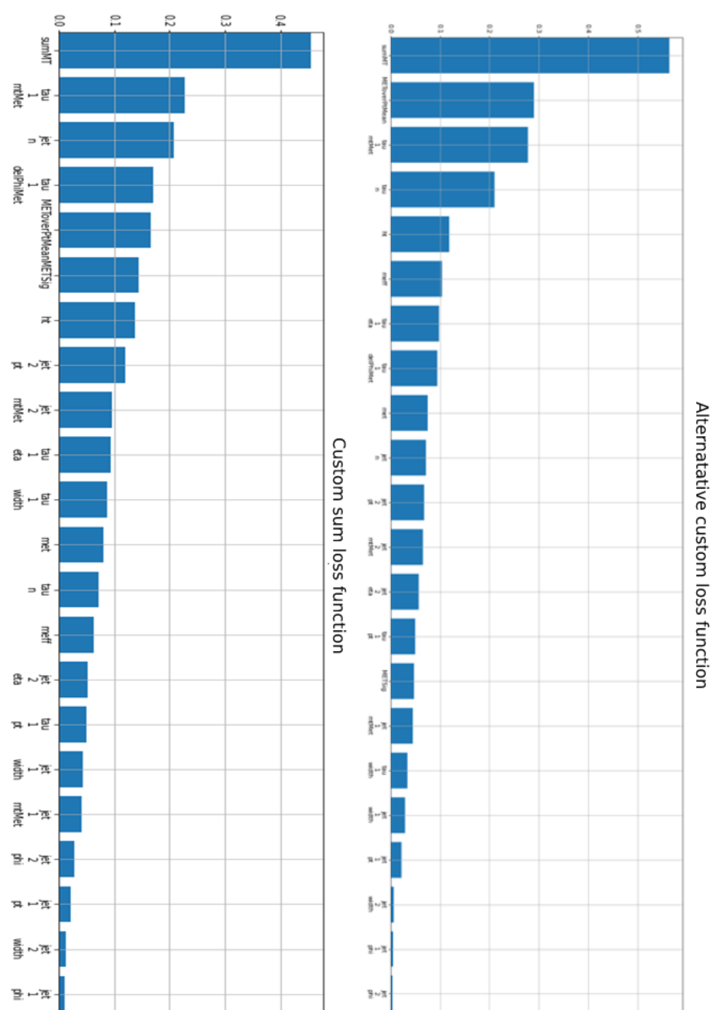


Figure 65: Feature distribution comparison of custom *sum* loss and alternative loss functions

8.7 Different signal files

During previous tests, the signal file in use has been *GG_**1100**_968_901_835.MC16a.hdf5*. However, the dataset used in this thesis contains two additional signal files: *GG_**1700**_1418_1276_1135.MC16a* and *GG_**2200**_1143_614_85.MC16a*. The performance of custom loss function and variance reduction was also tested using these files.

The first, different signal file to be tested was *GG_1700_1418_1276_1135.MC16a*. The test itself was akin to the previously performed tests with the usual signal file. For each of the loss function tested, several RF models were trained, and their performance was assessed using the same test datasets. The results of this test are summarized in the figure below.

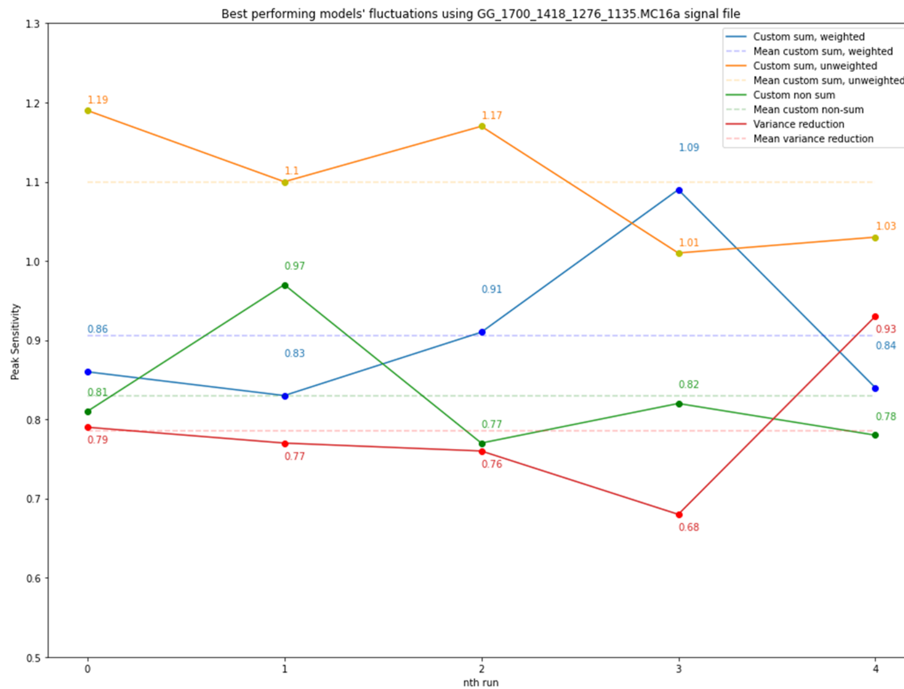


Figure 66: Deviations of peak sensitivities using GG_1700 signal file

A few noticeable differences from the standard signal file can be observed. First of all, the sensitivity values are much lower for all the models, despite the loss function used. While the sensitivities of the regular signal file were in the range of [5,7], the sensitivities of this signal file barely surpassed 1. The initial hypothesis was that there were not enough signal files in the train dataset for the model to learn the data patterns and differences between signal and background files. To test this hypothesis, few tests were performed.

The first one was to test the number of data entries in the different signal files. The typical signal file, *GG_1100* contained 5297 data entries, *GG_1700*

contained 3352 and *GG_2200* contained 4781 signal entries. This means that the GG_1700 signal file contained 37% less data than the GG_1100 file. This fact alone can have a severe impact on the models' performance, considering the already low signal to background ratio, however, such a drastic decrease in the performance in unanticipated due to this reason alone.

The second test involved making predictions on the same dataset the model was trained on, in addition to regular test dataset predictions. Predicting on the same dataset that the model was trained on provides an insight regarding whether the model underfits or overfits the data. Overfitting was not anticipated, however, if it did occur for some unexpected reason, it could explain the major difference in the sensitivities using different signal files. Nonetheless, as seen from Table 23, there were no significant differences between test and train datasets. This result indicated that no unintentional overfitting was prominent in the models.

	Test data sensitivity	**Train data sensitivity**
Sum custom with weighted voting	0.82	0.89
Sum custom with unweighted voting	1.16	1.17
Non-sum custom loss function	0.81	0.77
Weighted variance reduction	0.70	0.64

Table 23: Model predictions with test and train datasets

The third test performed included an uneven split of background and signal data in train/test datasets with adjusted the sampling weights. With uneven signal to background ratios in the train and test datasets, it is possible to increase the amount of signal data in the train dataset, which allows the ML models to better understand its patterns. At the same time, the decrease of signal amount in the test dataset is minimized with the adjusted sampling weights. Moreover, this method should also account for the difference in the number of data entries in the signal files and diminish its impact on the performance. The results of this test are summarized in the Figure 67.

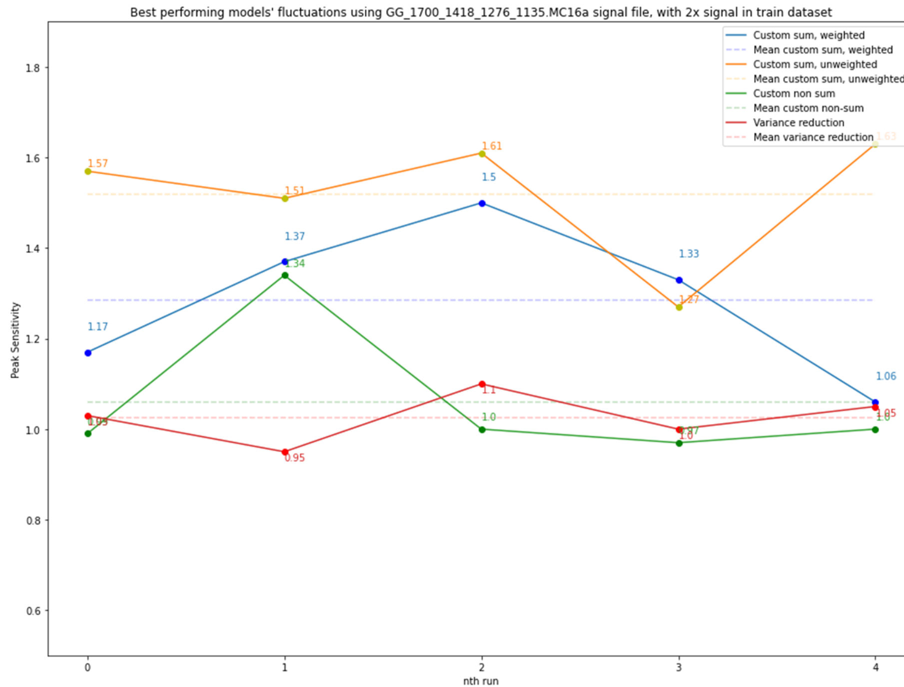


Figure 67: Deviations of peak sensitivities using GG_1700 signal file using unbalanced dataset

While uneven data splits did have an increase in the sensitivities, in comparison to the models' performance using regular data split, there was still a major difference between the signal files, which is not expected from relatively similar signal files.

The last considered option was a potential difference in the weights between the files. To test whether weights can account for this difference, the product of weights has been calculated for each of the data entry. The resulting weight has then been summed for each of the signal file, which ultimately represents their weighted count. The regular signal file, *GG_1100* had the weighted count of 946.41. In comparison, *the GG_1700* signal file had a weighted count of merely 61.84, despite having a relatively similar of unweighted count.

The large difference in weights explains the difference in peak sensitivities. If the weights are low, then the absolute amount of predicted signal and background data entries will be low, which ultimately results into low sensitivities. To address this issue, it is possible to use unweighted counts for sensitivity calculations. However, this solution would not provide meaningful representation of data, due to the importance of weights.

Regardless of the absolute difference in the peak sensitivities achieved between the different signal files, a comparative analysis of different loss functions can still be performed. Based on Figure 66 and Figure 67, similar trends can

be noticed in the relative performance between the different loss functions that were also prominent in models using the regular signal file. The model with the highest sensitivity achieved is the one using custom *sum* loss function with unweighted voting. Based on these figures, and comparing the performance of variance reduction and custom *sum* loss functions over multiple runs, it would be unlikely that such a difference could be entirely achieved by deviations. Especially considering that the worst performing RF using custom *sum* loss function was able to outperform the best model using variance reduction.

8.8 Performance using data from different data collection periods

During the previous tests, the data used in models' training and testing has been from data collection period ***a***. Performance and fluctuations tests have also been performed using a set of data files from different data collection periods - ***d*** and ***e***.

The process of performance and fluctuation testing has been performed the same way as described in the previous sections, with the exception of using different datasets.

8.8.1 Data Collection Period *d*

Apart from models performance using variance reduction loss function, the results using ***d*** data files were similar to the results using regular, ***a*** data files. The best performing models were the classification models using custom loss function, having the peak sensitivity of 6.88. Models using variance reduction, on the other hand, severely under-performed in comparison to their performance using *a* data files. Not only was the peak sensitivity lower (5.35 as opposed to 5.98), the fluctuations were also unexpectedly large. The difference in worst and best performing models using variance reduction with this dataset was as large as 2.11, whereas in the *a* dataset it was approximately 0.70.

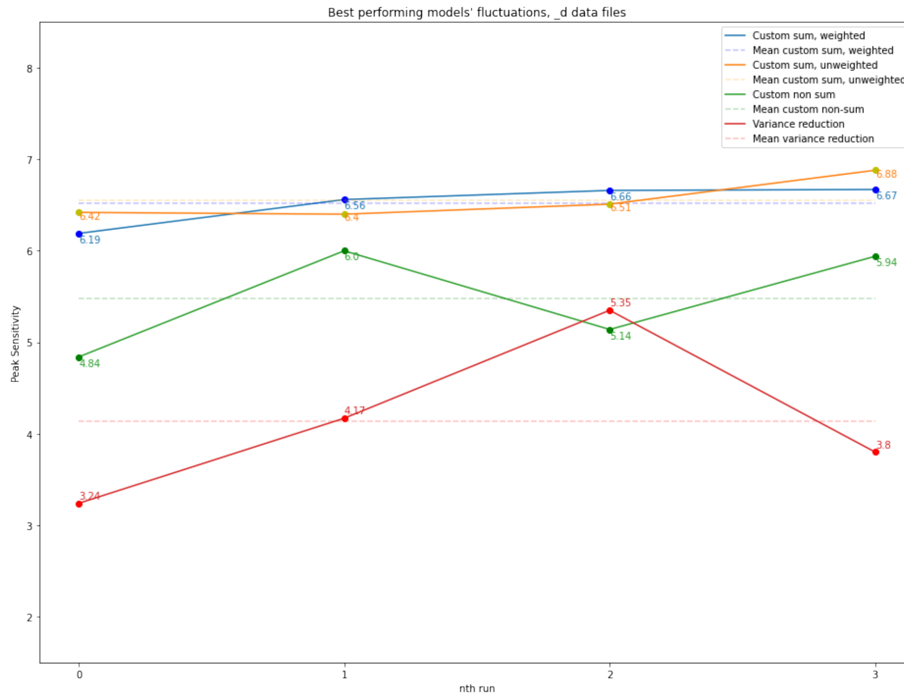


Figure 68: Deviations of peak sensitivities using d dataset

8.8.2 Data Collection Period e

As seen in Figure 69, the relative ML model performance using data from data-collection period $\boldsymbol{e}$ remains similar as in data taken from periods $\boldsymbol{a}$ and $\boldsymbol{d}$. ML models using the custom *sum* loss function tend to perform best, with an approximate 13% increase in peak sensitivity achieved as compared to variance reduction and *non-sum* custom loss function. Similarly as with data from periods $\boldsymbol{a}$ and $\boldsymbol{d}$, the variance reduction performed slightly worse than the *non-sum* custom loss function. However, the difference in peak sensitivities achieved between these two loss functions was not to the extent where it could not be explained due to fluctuations.

A general increase in sensitivities achieved using data from $\boldsymbol{e}$ collection period can also be observed. However, this is accounted to this data period having a higher signal to background ratio.

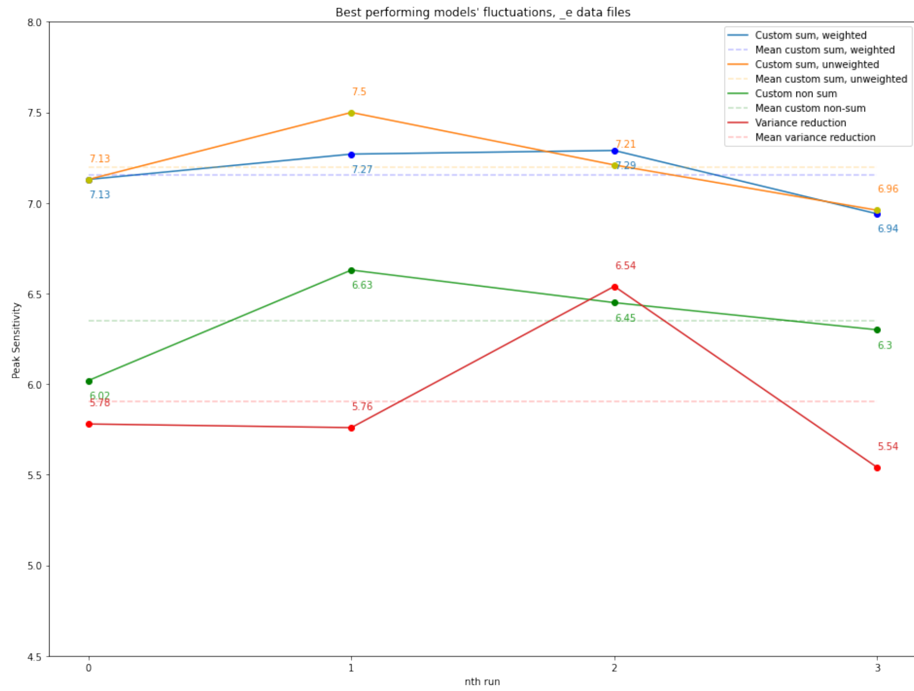


Figure 69: Deviations of peak sensitivities using e dataset

8.8.3 Remarks

The general performance was in align with the expectations between the data from all data collection periods, and similar performance trends were observed. Likewise, there were no noticeable differences in the feature importance distributions between the difference difference data collection periods.

8.9 Errors

Although accounting for errors is not strictly necessary in a high-level overview testing, it becomes important when the final conclusions are made regarding the results of our models and thereby loss functions used. The Equation (7) and Equation (8) describe how the errors were calculated, by using error propagation.

$$
\begin{aligned}
z &= \frac{s}{\sqrt{s+b}}, \\
\delta_z^2 &= (\frac{\partial z}{\partial s})^2 \delta_s^2 + (\frac{\partial z}{\partial b})^2 \delta_b^2 \\
&= \frac{1}{4(s+b)} \cdot s + \frac{s^2}{4(s+b)^3} \cdot b \\
&= \frac{s}{4(s+b)} [1 + \frac{sb}{4(s+b)^2}]
\end{aligned}
\tag{7}
$$

$$
\begin{aligned}
\delta_s &= \sqrt{s}, \delta_b = \sqrt{b} \\
\frac{\partial Z}{\partial s} &= \frac{\sqrt{s+b} - \frac{s}{2\sqrt{s+b}})}{s+b} \\
&= \frac{\frac{2(s+b)}{2\sqrt{s+b}} - \frac{s}{2\sqrt{s+b}}}{s+b} \\
&= \frac{s+b}{2(s+b)^{\frac{3}{2}}} \\
&= \frac{1}{2\sqrt{s+b}} \\
\frac{\partial Z}{\partial s} &= -\frac{s}{2(s+b)^{\frac{3}{2}}}
\end{aligned}
\tag{8}
$$

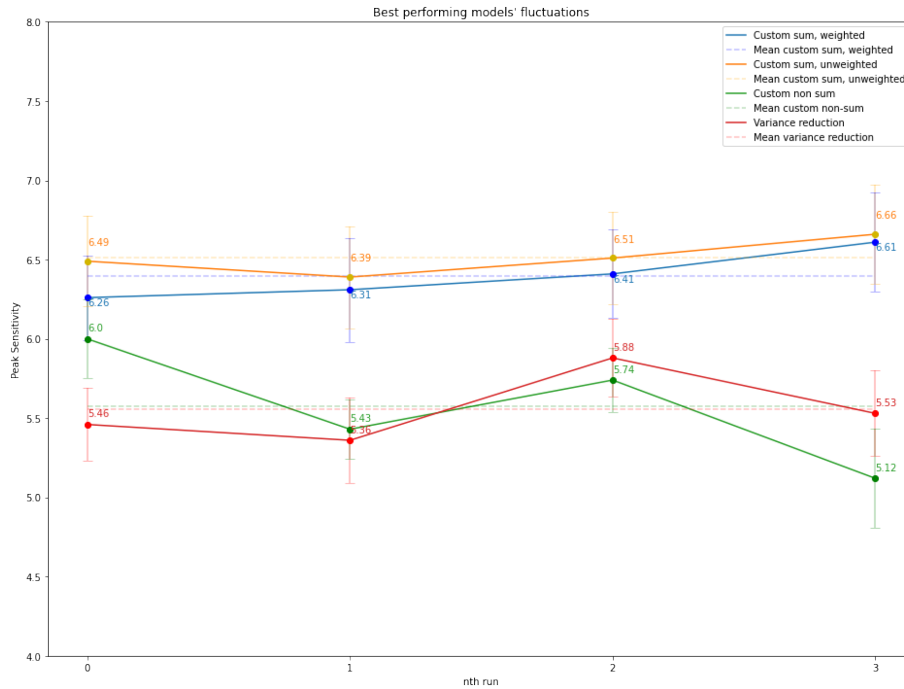


Figure 70: Deviations of peak sensitivities using a dataset, with error bars

Figure 70 is a representation of Figure 48 with error bars. For these particular results, the errors varied anywhere between 0.14 and 0.36, depending on the run and the loss function used.

Let us select the best performing models for each of the loss function assessed. In this case, the model with custom loss function would have the sensitivity of 6.66 and the model using variance reduction would have the sensitivity of 5.88. If we assume the worst case scenario for the custom loss function, in which its error value (0.31) is subtracted from its peak sensitivity, and the error value of the best performing variance model (0.24) is added to its sensitivity, there still is a difference of 0.23 in the sensitivity, in the favour of model using the custom loss function. This fact further increase the confidence regarding the performance of custom loss functions.

8.10 Unbalanced datasets

From machine learning's perspective, one of the main reasons why the problem of separating signal and background is so difficult is due to the fact the signal to background ratio is extremely low. One way to tackle this issue, which has also been described in Section 8.7, is to increase the amount of signal data in the training dataset, at the expense of reducing its quantity in the test dataset.

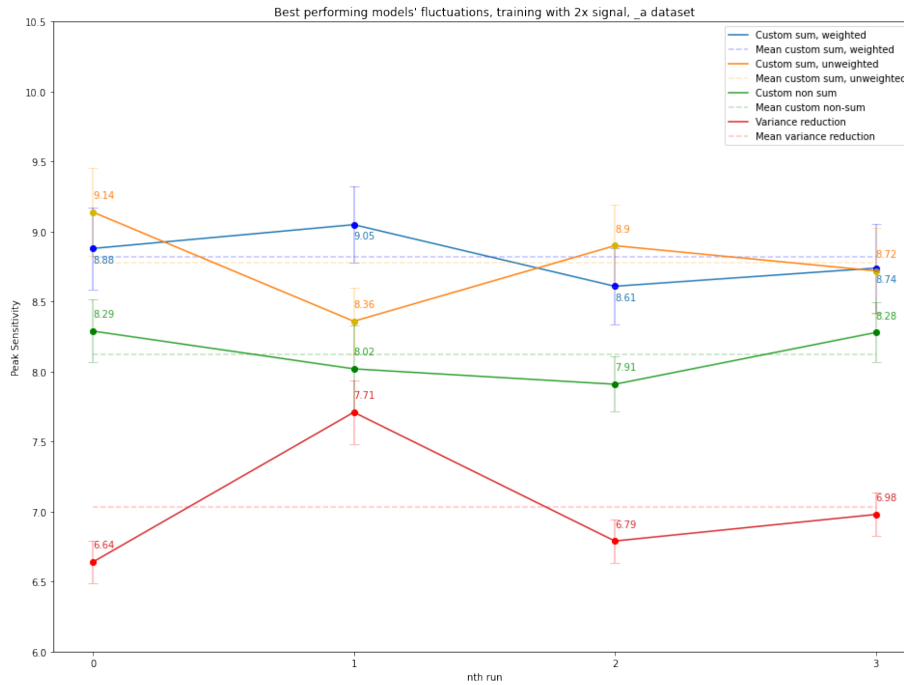


Figure 71: Deviations of peak sensitivities using a dataset and increased signal in the training data, with error bars

The figure above provides similar information as Figure 70, the only difference being that the dataset, which all of the models have been trained on, has had its signal ratio increased by the factor of 2, as compared to its test dataset, while at the same time adjusting their sampling weights. The following differences can be observed using this approach:

1. The sensitivity of the best performing model using the custom loss function has increased to 9.14 from 6.66.
2. The sensitivity of the best performing model using variance reduction as its function has increased to 7.71 from 5.88.
3. The difference in sensitivities between variance reduction and the custom loss function has increased from 0.78 to 1.43.
4. The difference between the non-sum custom loss function and the variance reduction became more prominent. While there was barely any difference in the regular tests between these two loss functions, with an increased signal in the training dataset, the non-sum custom loss function outperformed variance reduction.

9 Conclusions

The key question this thesis attempted to answer was whether currently used ML models for separating background and signal data in high energy physics can be improved by applying a different, specialized loss function. Using the custom loss function, as described in the earlier parts of the thesis, the models' sensitivity increased by approximately 11%. Hence, based on these results, it can be concluded that it is indeed possible.

Nonetheless, even if a thorough testing was performed, taking into a consideration various factors affecting ML model's performance, there is always room for additional testing and improvement. In comparison to existing state-of-the-art machine learning libraries, certain restrictions affecting models' performance had to be implemented in the custom implementation of random forest and decision tree models. However, taking into a consideration that the restrictions did not have a severe impact on the performance, and that same restrains have been applied when comparing different loss functions, the underlying difference in the performance is unlikely to be caused by the said restrictions. Nonetheless, it is possible to rewrite the code in a runtime-efficient language, reducing, or even removing any constraints. In this thesis, this approach was deemed too time-consuming, as the intention was not to produce a new machine learning library, but to assess the potential of a new loss function.

Despite all of these factors, the custom loss function implemented in this thesis does have an undeniable potential, and it provides definitive results. Based on the results and observations made in this thesis, it can be concluded that, in specific scenarios, its ability to achieve sensitivity has a potential to surpass traditional loss functions used in random forest and decision tree models.

10 Further Work

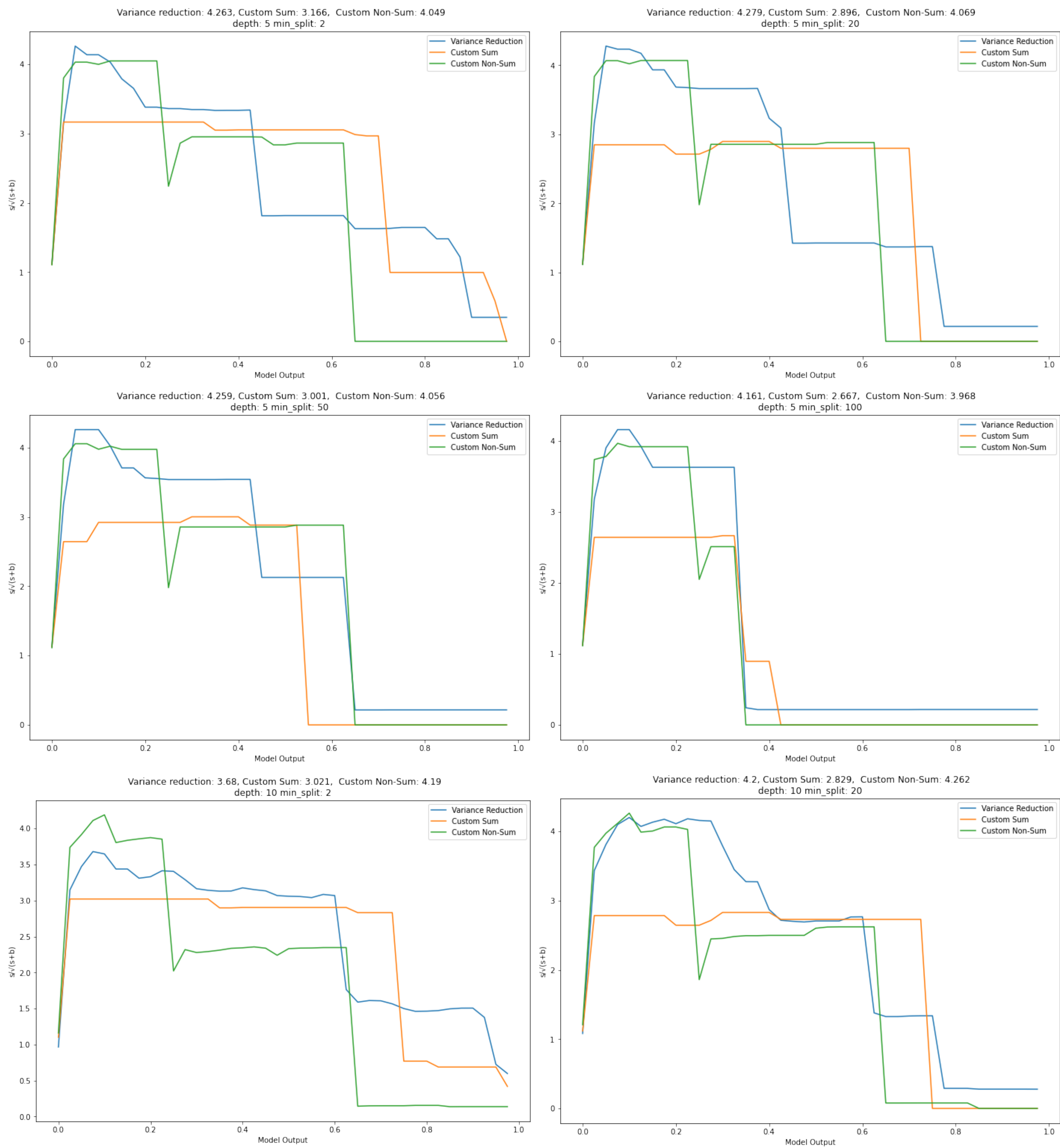
As mentioned previously, the implementation of decision trees and random forest, based on which the conclusions have been made, is sufficient, yet not ideal. Further testing would benefit greatly of a more optimized implementation. Based on the experience and research done in this thesis, the best possible approach to this problem would be a change in the choice of programming language, to one of a greater runtime-performance. Further testing can be divided into three general parts: continuing current analysis, verifying performance using a different, HEP-related, dataset and using a dataset from a non-physics related field.

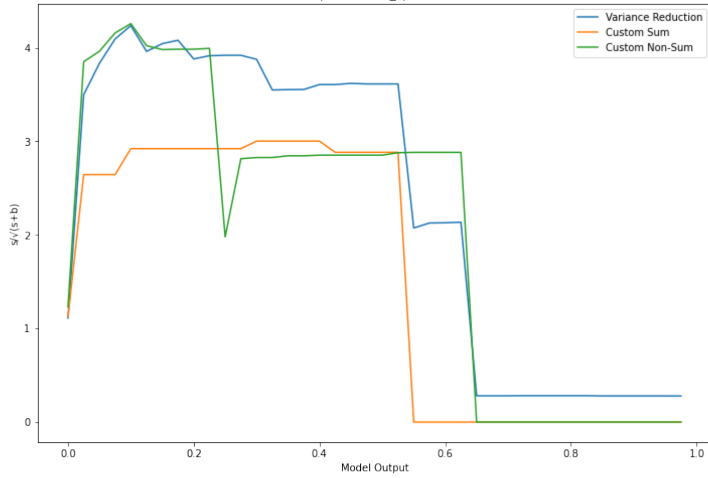
With a better runtime-performance, many of the already applied analysis methods could be performed at a greater depth. One of such methods would be to increase the number of estimators in random forest algorithms, without the reduction in data used per tree. In general, a better model runtime would provide a better possibility of finding hyperparameter combinations that most accurately allow different loss functions to achieve their best performance.

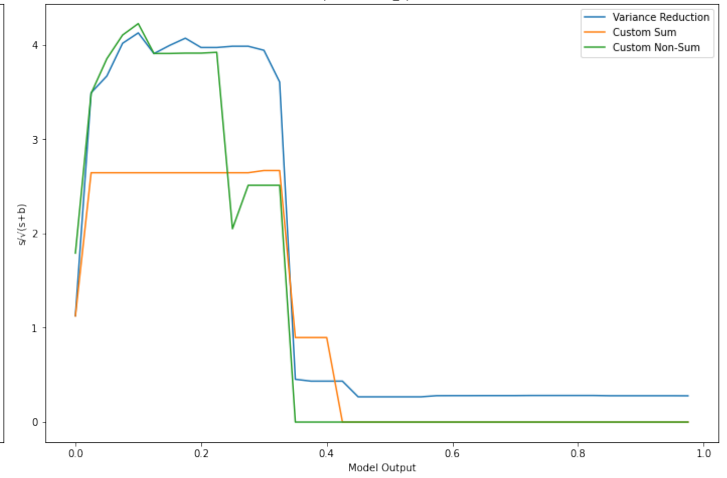
While there were significant improvements found using the custom loss function, they were not groundbreaking. There is a possibility that the loss function would be of use in particle research, however, further research is necessary to validate it. In this thesis, the comparison of custom loss function has mainly been performed by comparing it to the same machine learning algorithms using conventional loss functions. However, it is not guaranteed that tree-based algorithms provide the best performance in particle research. In an ideal scenario, the performance of the loss function would be compared with machine learning models using different algorithms and different datasets, which would provide a better insight into its overall performance in particle research. This includes different tree-based algorithms, such as XGBoost, which would also have the ability to use the custom loss function, but also performance comparisons with non-tree-based algorithms, to test its the general viability.

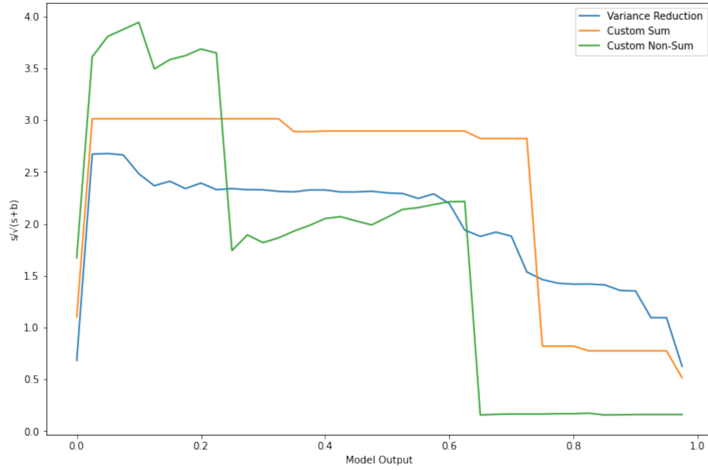
Moreover, the generality of this particular loss function has not yet been tested. While the intention of the custom loss function was to achieve the highest sensitivity in HEP-related problems, it is also possible to apply the same principles in other binary classification problems, where the goal is to achieve the highest sensitivity possible.

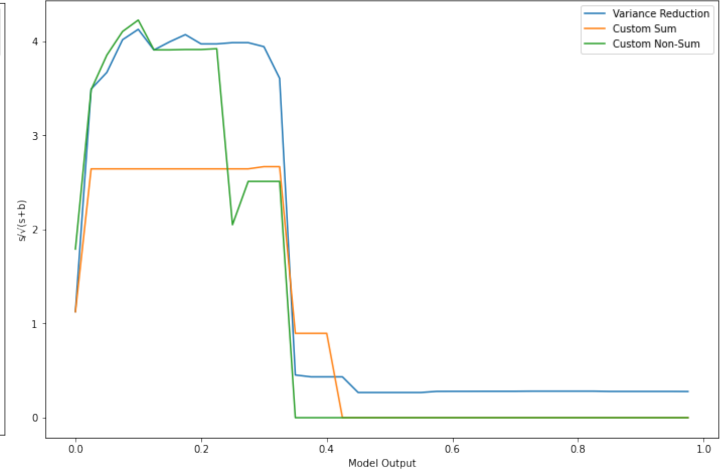
A Decision tree hyperparameter testing

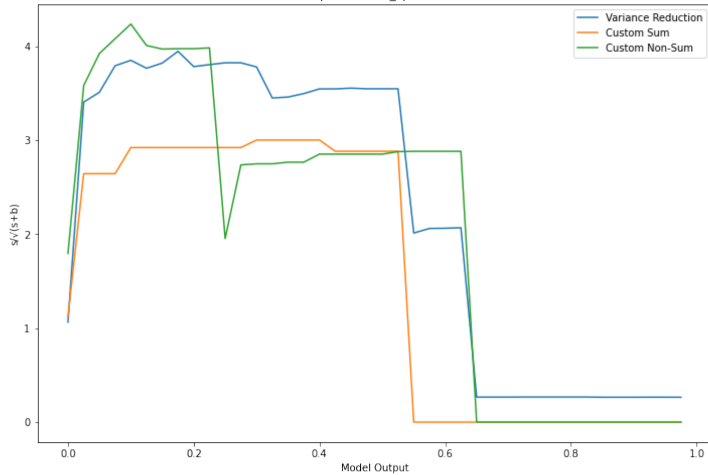


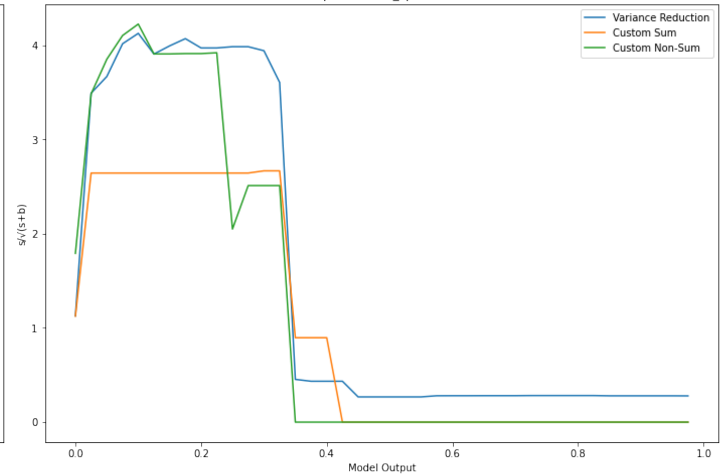

Variance reduction: 4.235, Custom Sum: 3.001, Custom Non-Sum: 4.256
depth: 10 min_split: 50
Variance Reduction
Custom Sum
Custom Non-Sum
s/√(s+b)
Model Output


Variance reduction: 4.126, Custom Sum: 2.667, Custom Non-Sum: 4.224
depth: 15 min_split: 100
Variance Reduction
Custom Sum
Custom Non-Sum
s/√(s+b)
Model Output

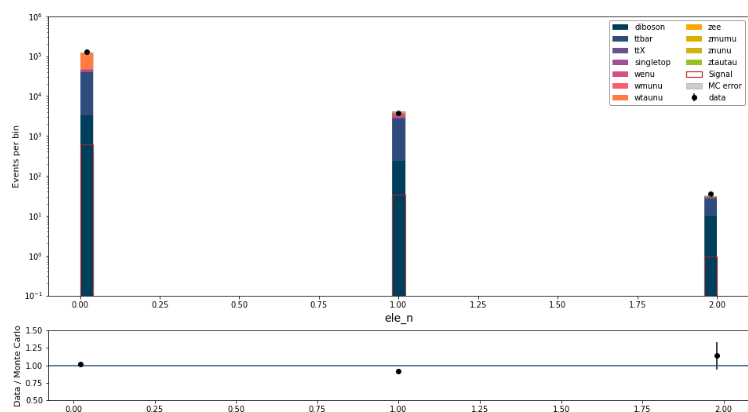

Variance reduction: 2.677, Custom Sum: 3.014, Custom Non-Sum: 3.944
depth: 15 min_split: 2
Variance Reduction
Custom Sum
Custom Non-Sum
s/√(s+b)
Model Output

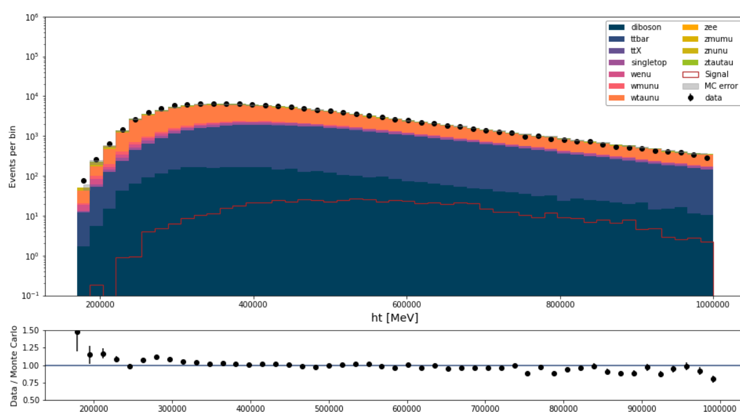

Variance reduction: 4.126, Custom Sum: 2.667, Custom Non-Sum: 4.224
depth: 15 min_split: 100
Variance Reduction
Custom Sum
Custom Non-Sum
s/√(s+b)
Model Output

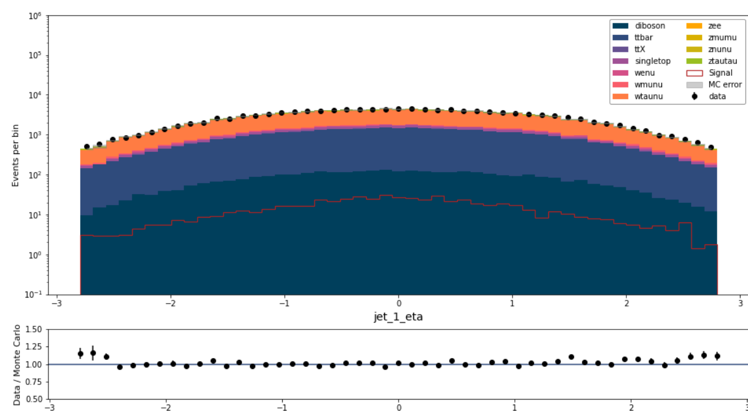

Variance reduction: 3.945, Custom Sum: 3.001, Custom Non-Sum: 4.235
depth: 15 min_split: 50
Variance Reduction
Custom Sum
Custom Non-Sum
s/√(s+b)
Model Output

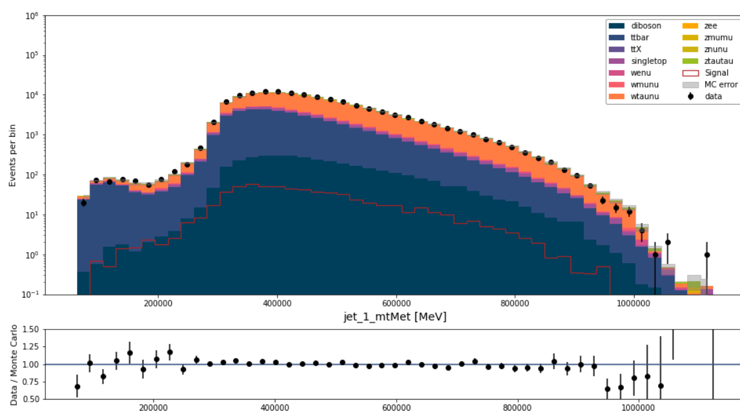

Variance reduction: 4.126, Custom Sum: 2.667, Custom Non-Sum: 4.224
depth: 15 min_split: 100
Variance Reduction
Custom Sum
Custom Non-Sum
s/√(s+b)
Model Output

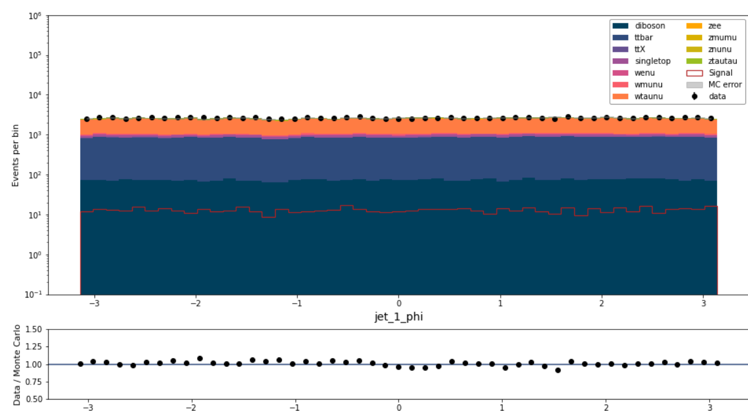
B Feature validation plots

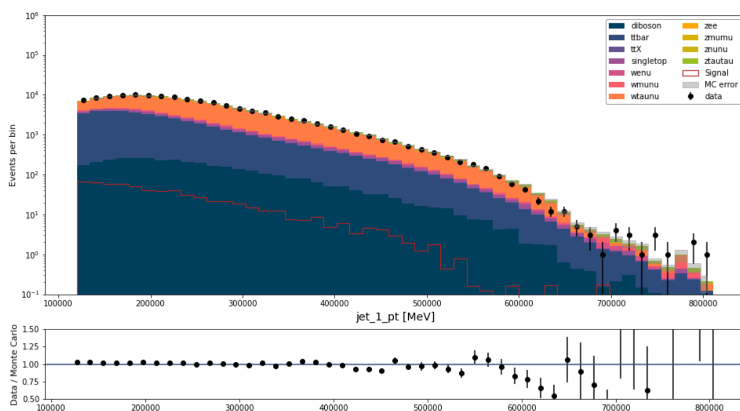


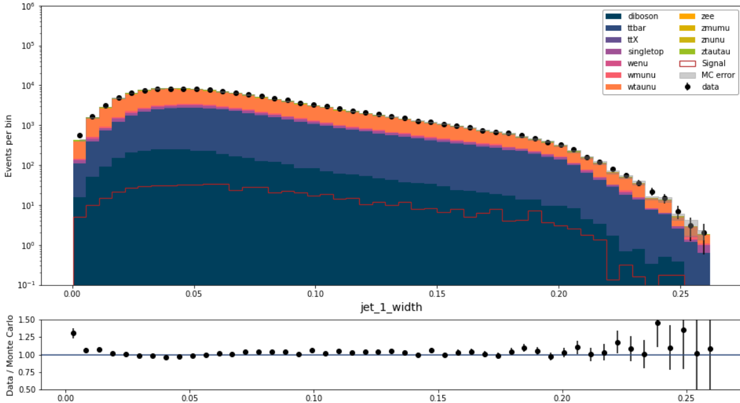


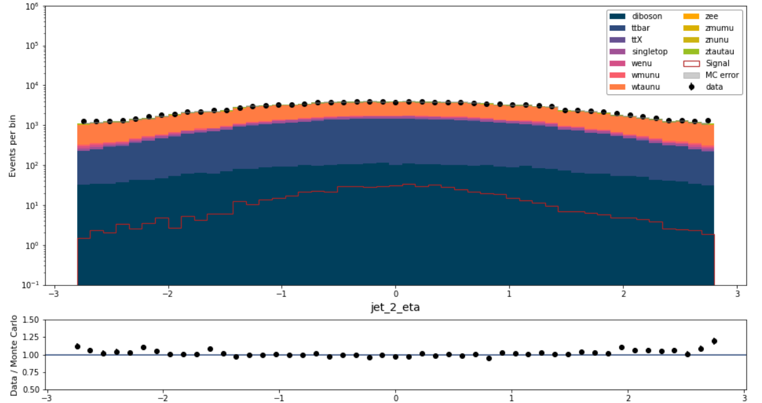


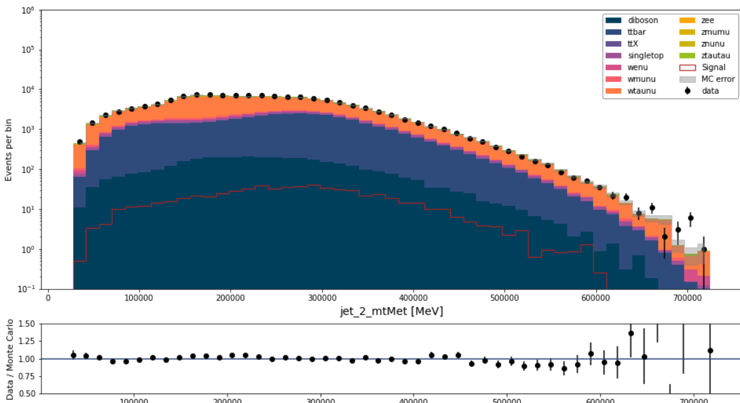


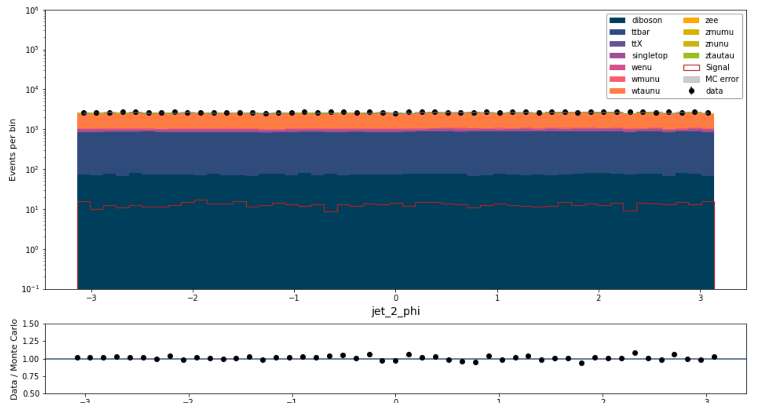


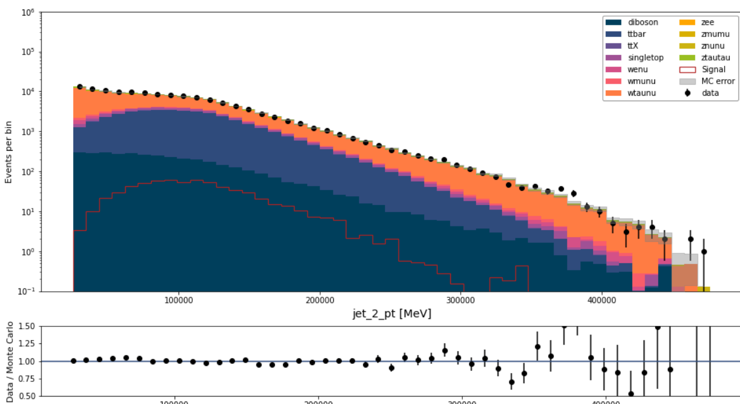


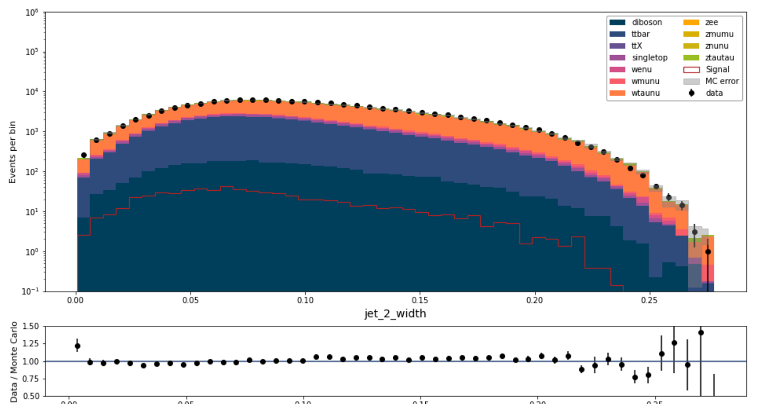

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
jet_1_width
Data / Monte Carlo

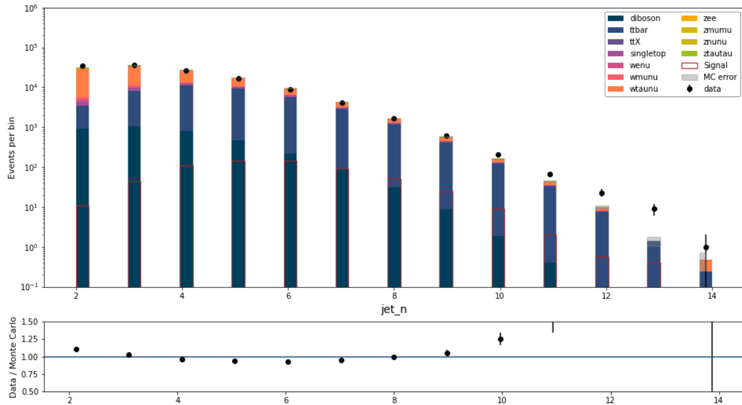

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
jet_2_eta
Data / Monte Carlo

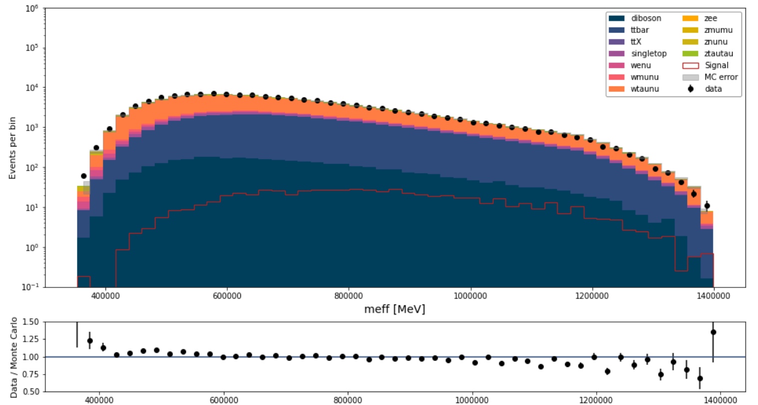

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
jet_2_mtMet [MeV]
Data / Monte Carlo

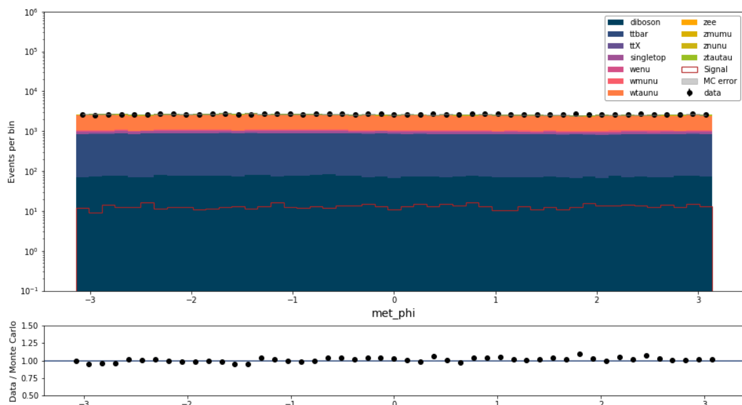

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
jet_2_phi
Data / Monte Carlo

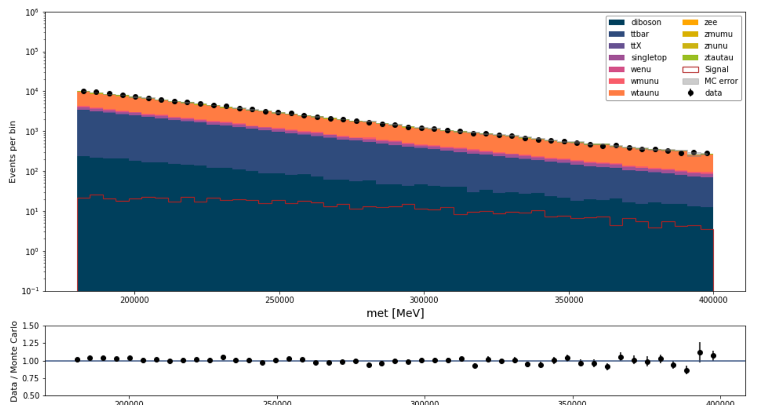

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
jet_2_pt [MeV]
Data / Monte Carlo

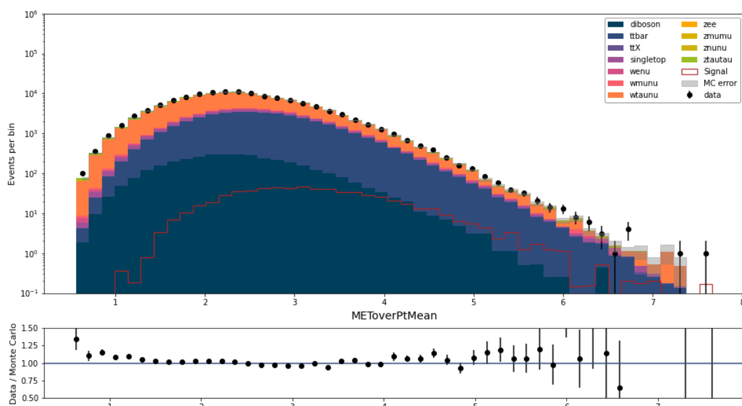

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
jet_2_width
Data / Monte Carlo

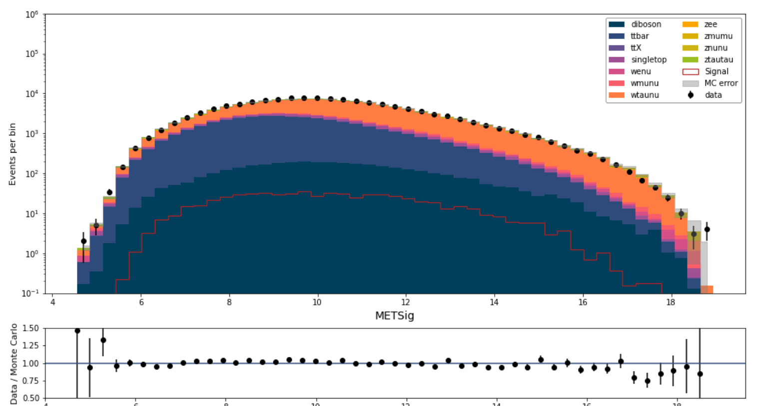

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
jet_n
Data / Monte Carlo

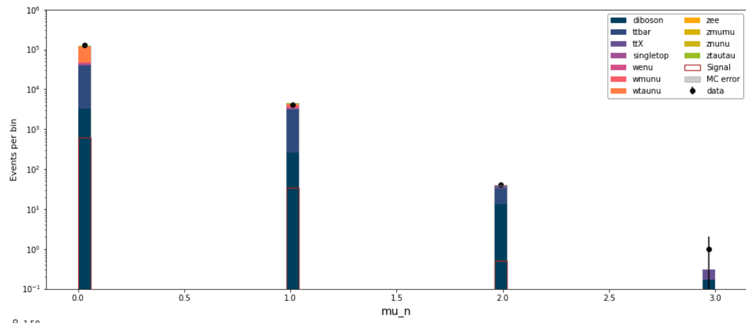

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
meff [MeV]
Data / Monte Carlo


diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
met_phi
Data / Monte Carlo

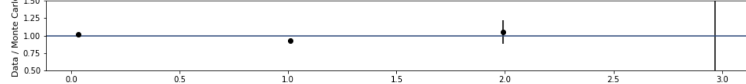

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
met [MeV]
Data / Monte Carlo


diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
METoverPtMean
Data / Monte Carlo

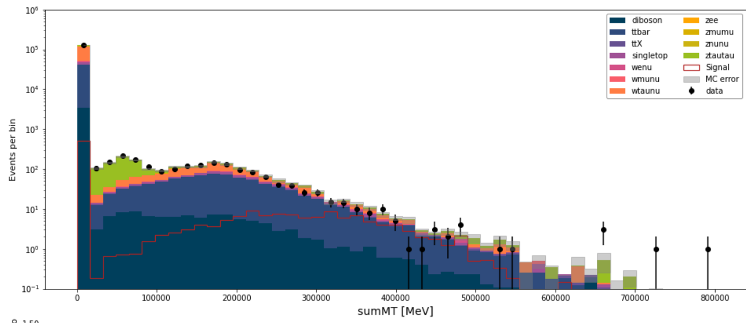

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
METSig
Data / Monte Carlo



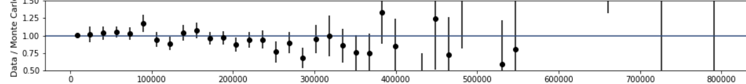
diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
mu_n



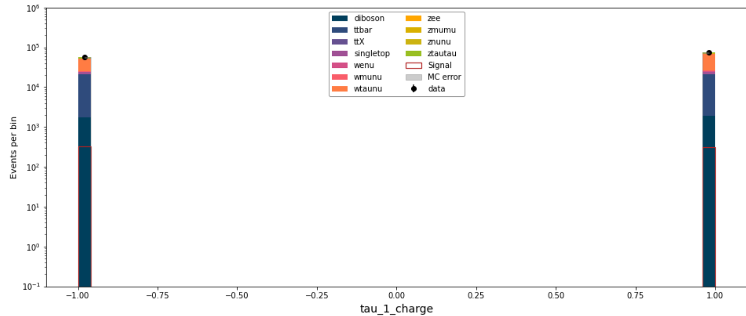
Data / Monte Carlo



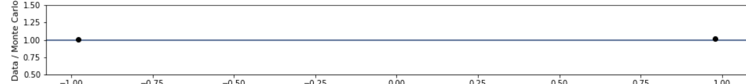
diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
sumMT [MeV]



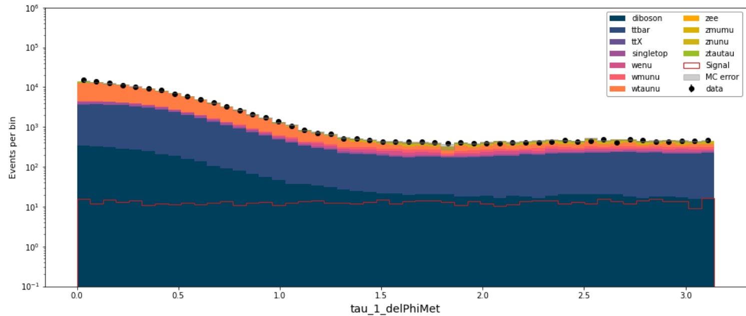
Data / Monte Carlo



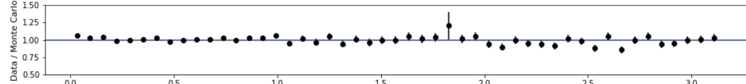
diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_charge



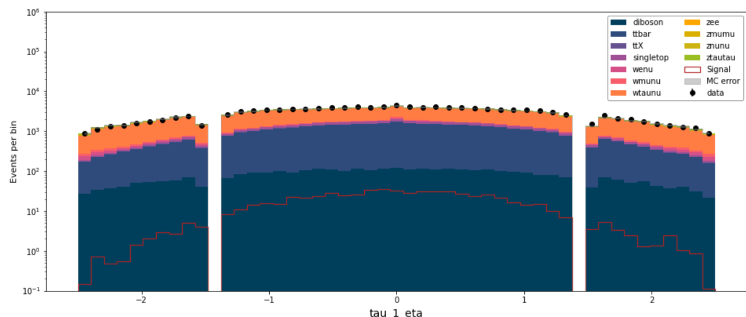
Data / Monte Carlo



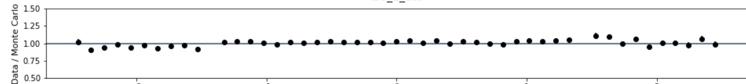
diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_delPhiMet



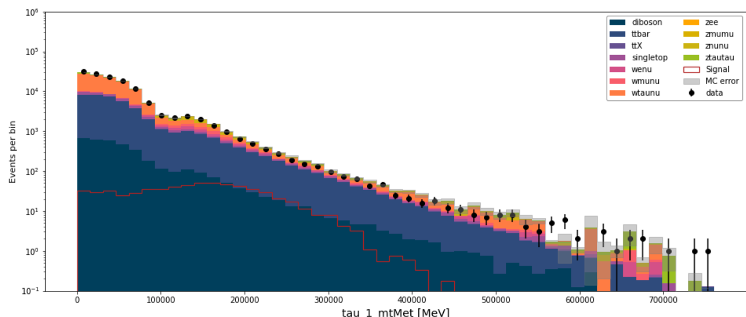
Data / Monte Carlo



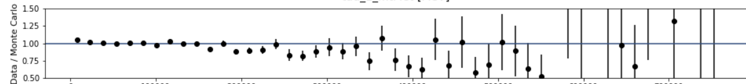
diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_eta



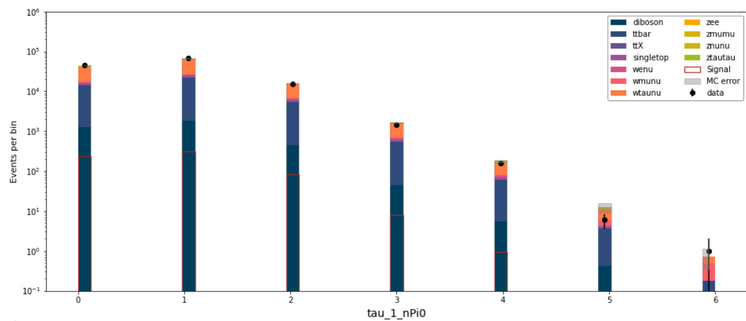
Data / Monte Carlo

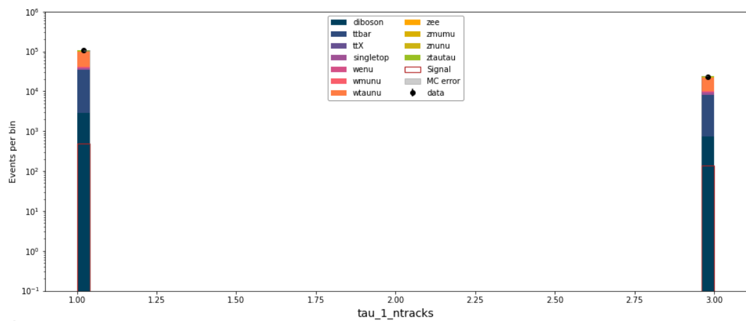


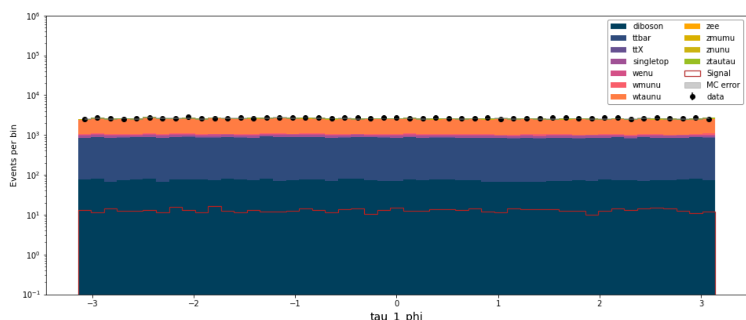
diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_mtMet [MeV]

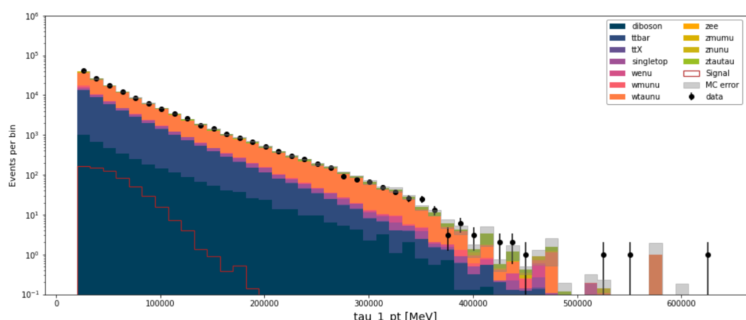


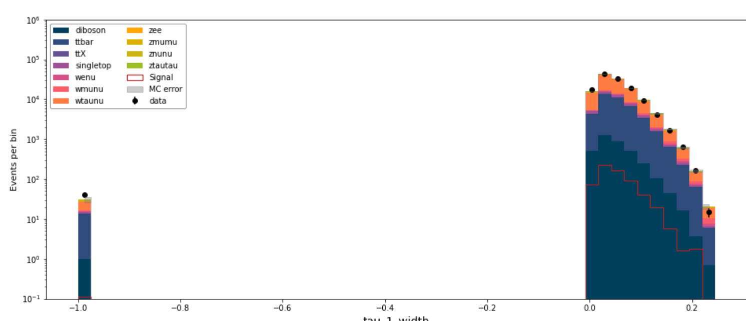
Data / Monte Carlo

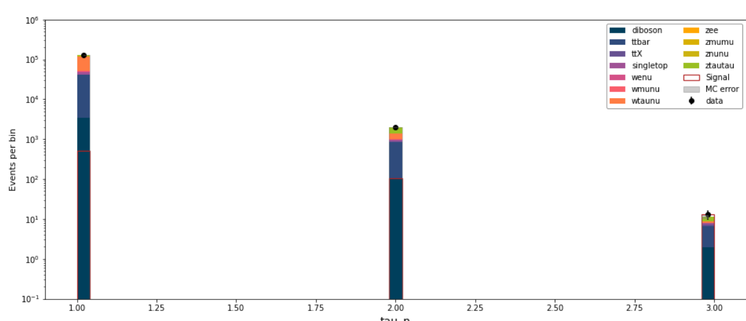

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_nPi0
Data / Monte Carlo


diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_ntracks
Data / Monte Carlo

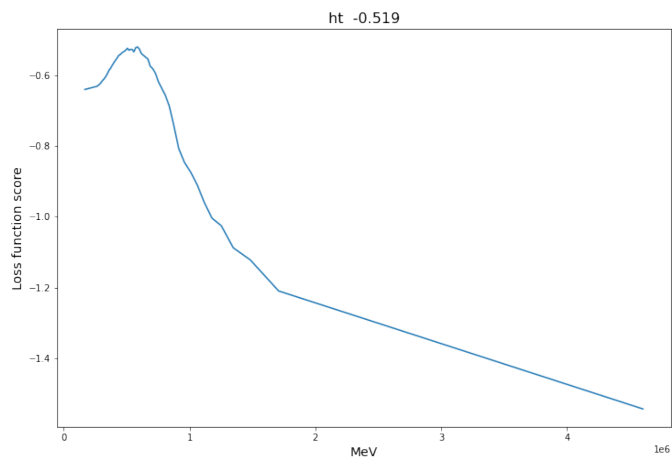

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_phi
Data / Monte Carlo

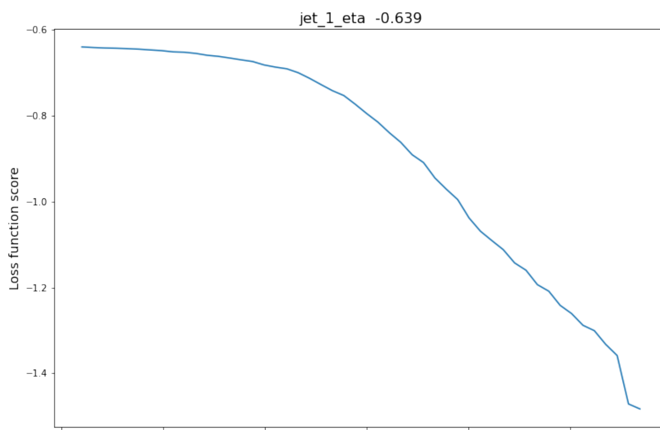

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_pt [MeV]
Data / Monte Carlo

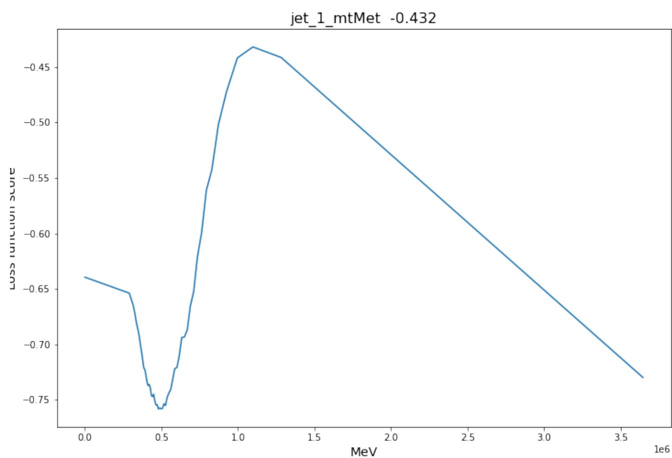

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_1_width
Data / Monte Carlo

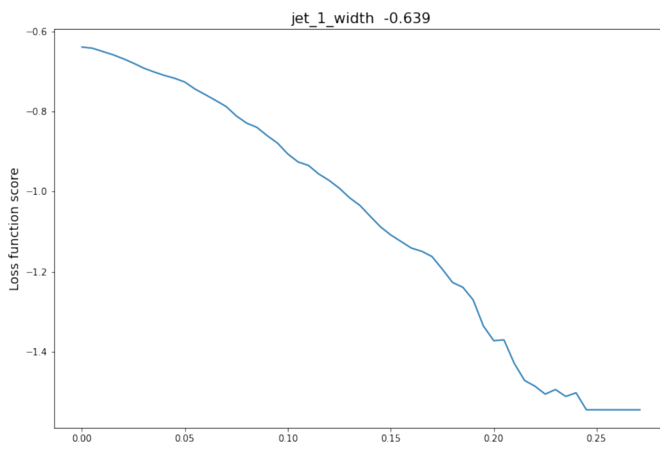

diboson
ttbar
ttX
singletop
wenu
wmunu
wtaunu
zee
zmumu
znunu
ztautau
Signal
MC error
data
Events per bin
tau_n
Data / Monte Carlo

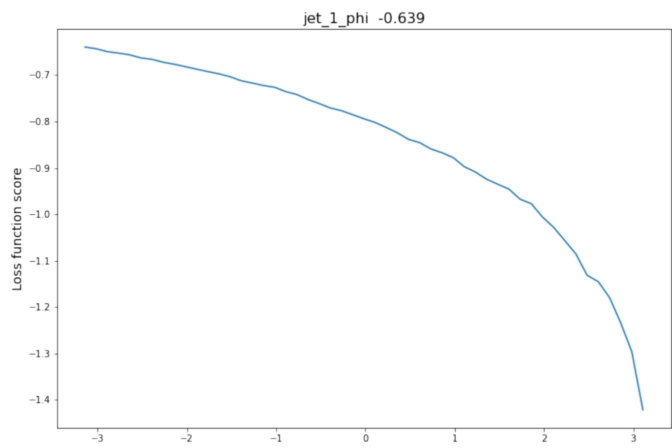
C Custom loss split verification

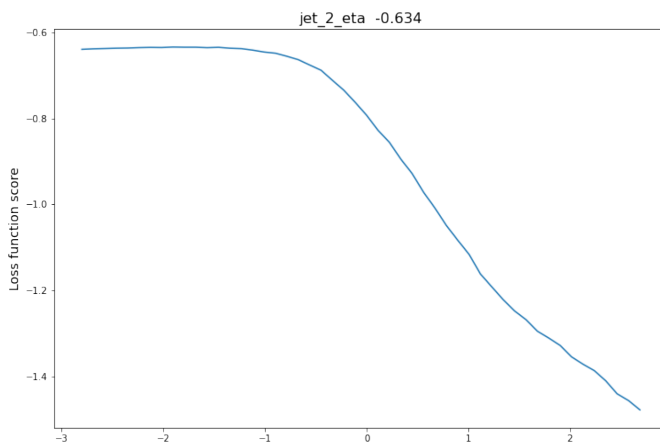


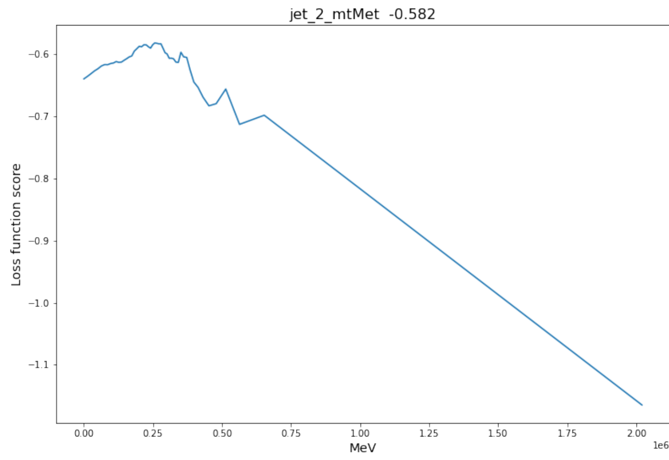


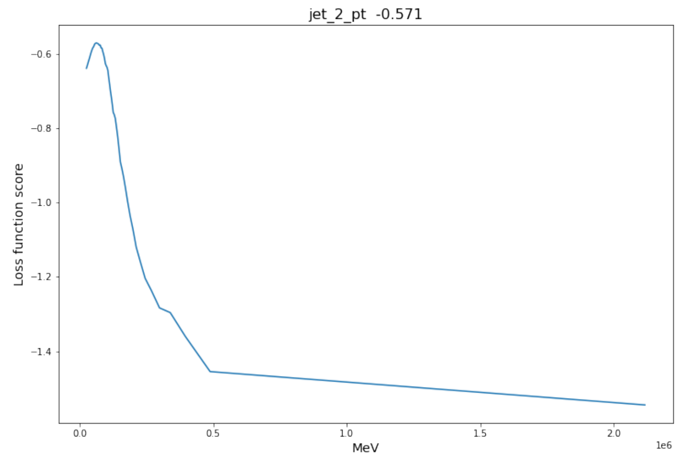


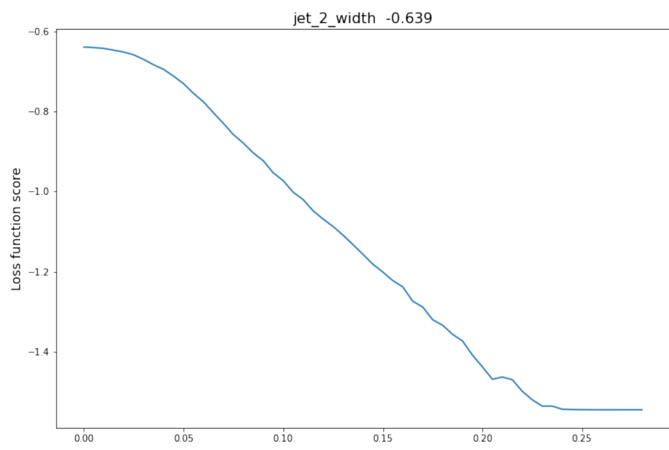


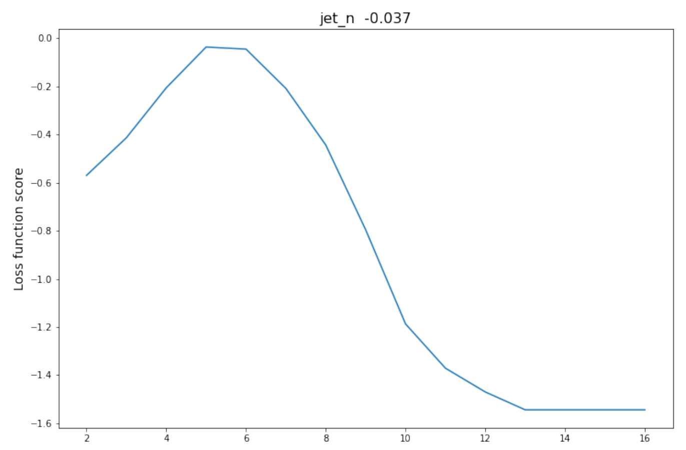


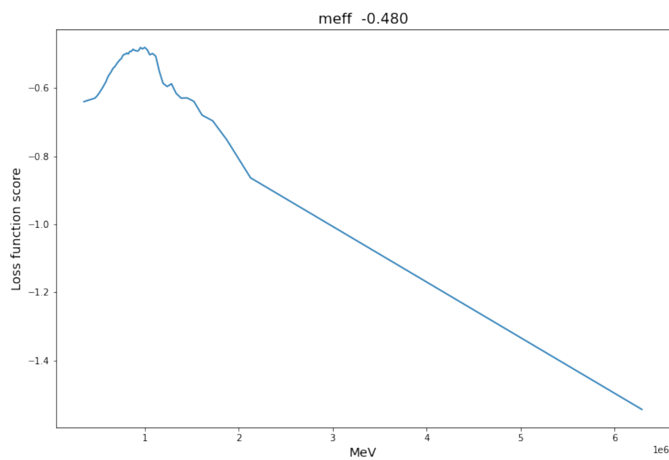


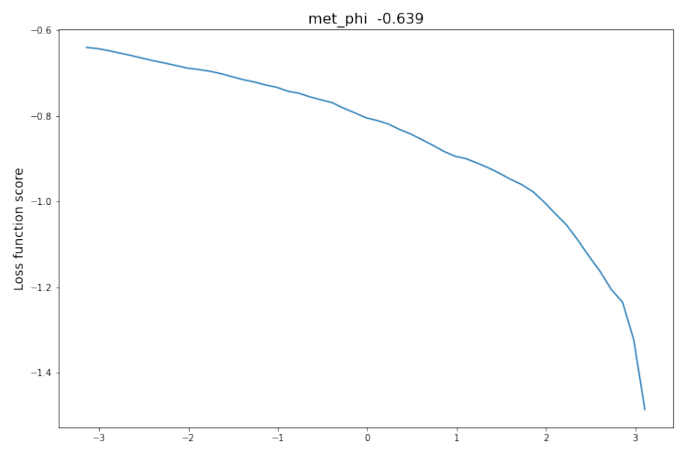

jet_2_mtMet -0.582
Loss function score
MeV
1e6

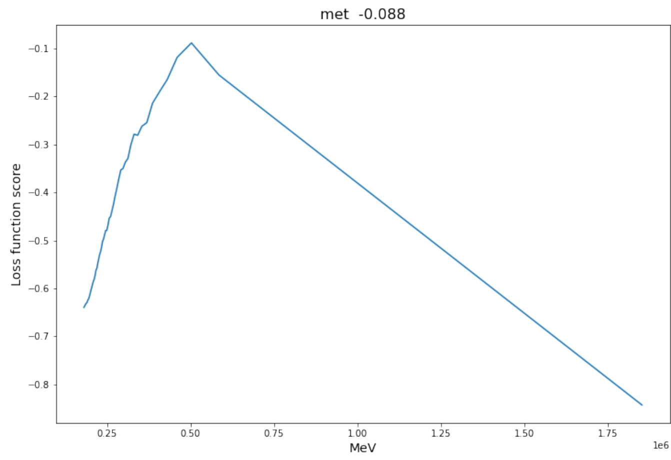

jet_2_pt -0.571
Loss function score
MeV
1e6


jet_2_width -0.639
Loss function score

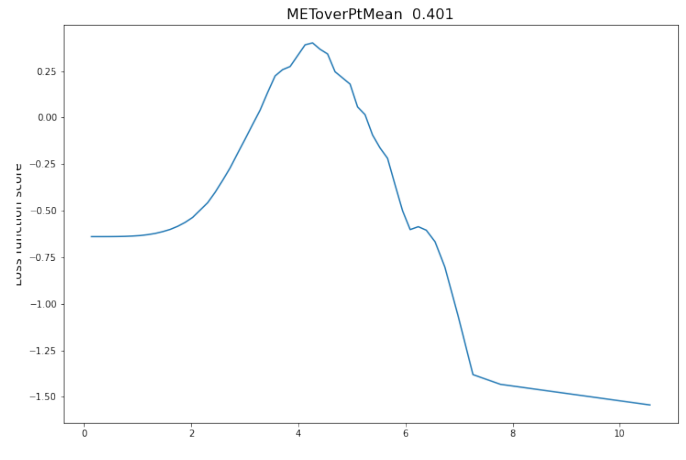

jet_n -0.037
Loss function score


meff -0.480
Loss function score
MeV
1e6

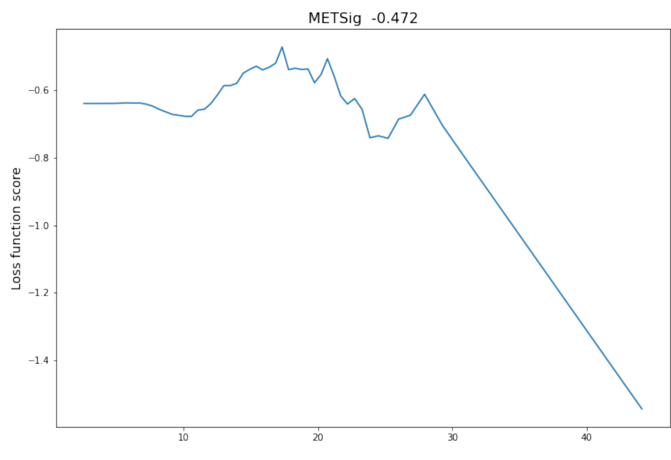

met_phi -0.639
Loss function score



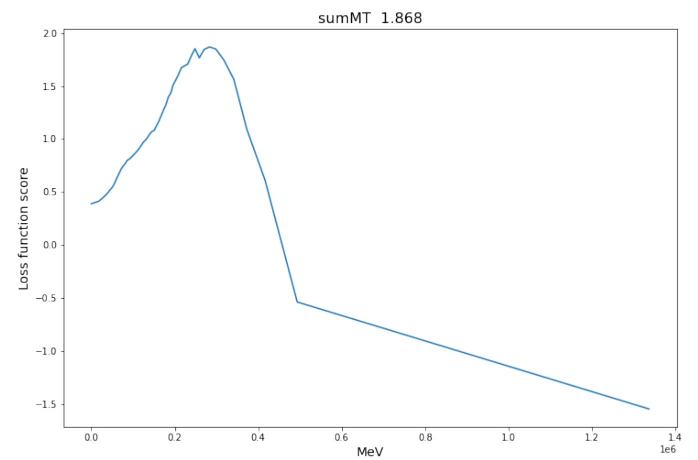
met -0.088
Loss function score
MeV
1e6
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
−0.7
−0.8
0.25
0.50
0.75
1.00
1.25
1.50
1.75



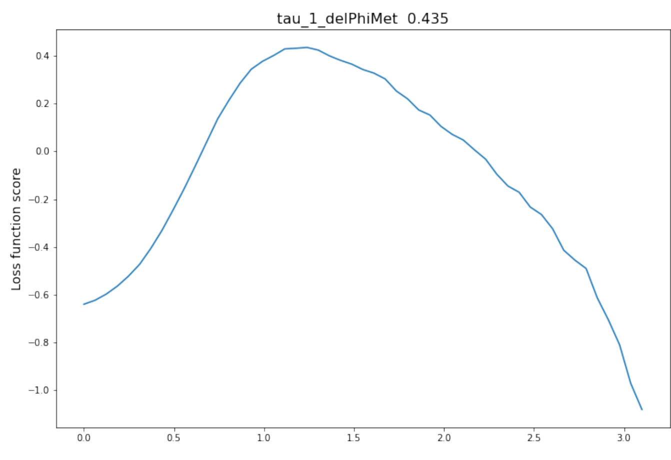
METoverPtMean 0.401
Loss function score
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
0
2
4
6
8
10



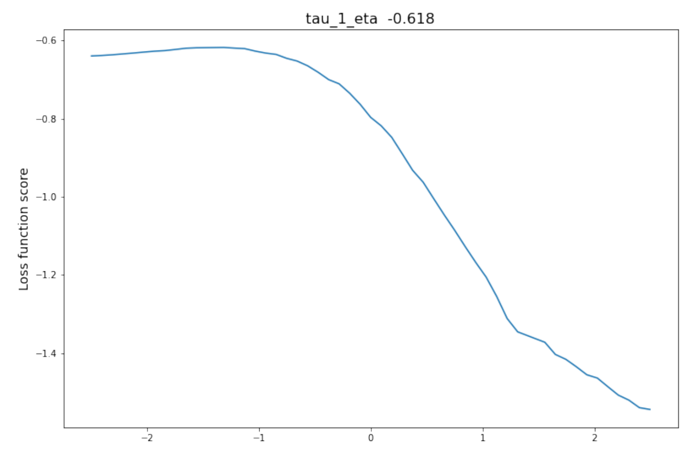
METSig -0.472
Loss function score
−0.6
−0.8
−1.0
−1.2
−1.4
10
20
30
40



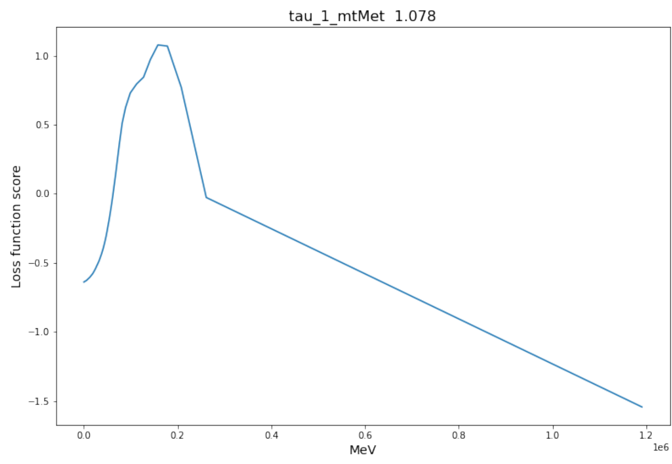
sumMT 1.868
Loss function score
MeV
1e6
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

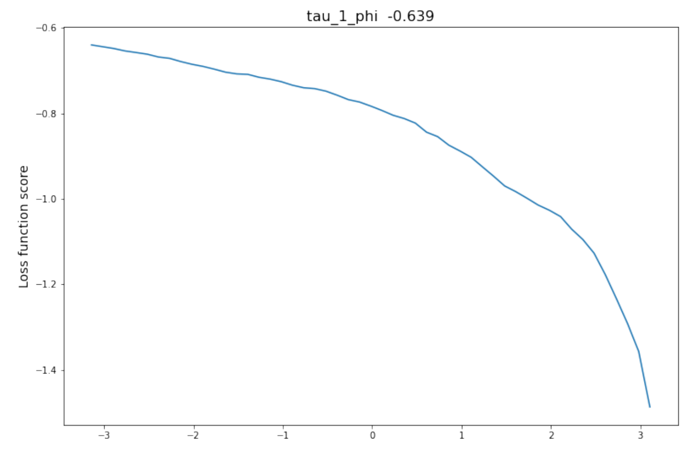


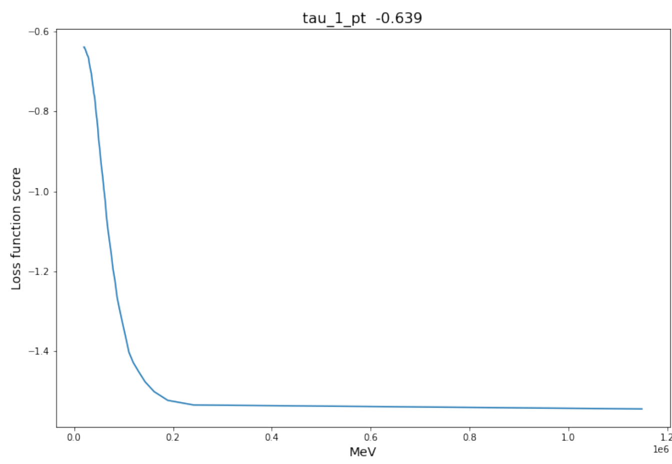
tau_1_delPhiMet 0.435
Loss function score
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

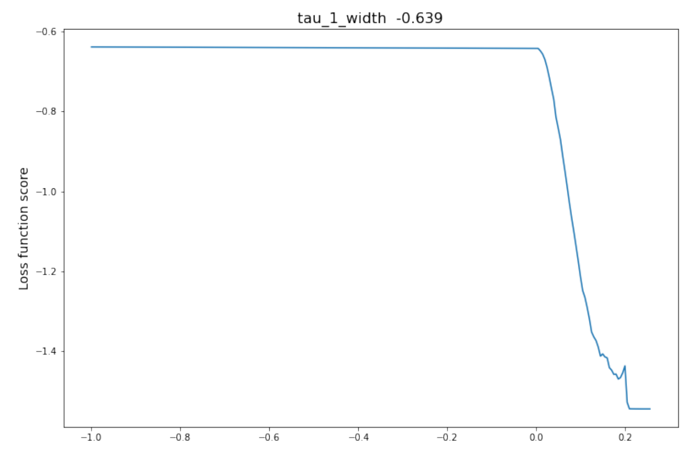


tau_1_eta -0.618
Loss function score
−0.6
−0.8
−1.0
−1.2
−1.4
−2
−1
0
1
2


tau_1_mtMet 1.078
Loss function score
MeV
1e6
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0
1.2


tau_1_phi -0.639
Loss function score
−0.6
−0.8
−1.0
−1.2
−1.4
−3
−2
−1
0
1
2
3


tau_1_pt -0.639
Loss function score
MeV
1e6
−0.6
−0.8
−1.0
−1.2
−1.4
0.0
0.2
0.4
0.6
0.8
1.0
1.2


tau_1_width -0.639
Loss function score
−0.6
−0.8
−1.0
−1.2
−1.4
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2

D Source code

The original notebook that has been in use throughout this project contains over 600 cells, involving a wide variety of tests. Since it uses data from the ATLAS experiment, the entirety of the notebook will not be attached to the thesis. The notebook, attached in the link below, contains all the code responsible for decision trees, random forest and generating model performance information, such as sensitivity plots. The majority of the code is explained in the Section 6.3.

Link to the source code:
https://github.com/dsp0011/MasterThesis/

References

[1] CERN. The Large Hadron Collider, 2021. URL https://home.cern/science/accelerators/large-hadron-collider. [Online; accessed October 17, 2021].

[2] CERN. The ATLAS Detector, 2021. URL https://atlas.cern/discover/detector. [Online; accessed October 17, 2021].

[3] CERN. Dark Matter, 2021. URL https://home.cern/science/physics/dark-matter. [Online; accessed October 17, 2021].

[4] Wikimedia Commons. Standard Model of Elementary Particles, 2019. URL https://en.wikipedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg. [Online; accessed June 01, 2021].

[5] CERN. The Standard Model, 2021. URL https://home.cern/science/physics/standard-model. [Online; accessed October 18, 2021].

[6] David Griffiths. *LaTeX: Introduction to Elementary Particles.* 2 edition, 2008.

[7] Palash B. Pal. *LaTeX: An Introductory Course of Particle Physics.* 1 edition, 2015.

[8] Identification of Jets Containing *b*-Hadrons with Recurrent Neural Networks at the ATLAS Experiment. Technical report, CERN, Geneva, Mar 2017. URL https://cds.cern.ch/record/2255226. All figures including auxiliary figures are available at https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/PUBNOTES/ATL-PHYS-PUB-2017-003.

[9] Adam Elwood and Dirk Krücker. Direct optimisation of the discovery significance when training neural networks to search for new physics in particle colliders. 2018. doi: https://arxiv.org/pdf/1806.00322.pdf.

[10] CERN. The Aerial View of the underground LHC, 2001. URL https://mediaarchive.cern.ch/MediaArchive/Photo/Public/2001/0107014/0107014_01/0107014_01-A5-at-72-dpi.jpg. [Online; accessed November 16, 2021].

[11] CERN. CERN's Accelerator Complex, 2013. URL http://cds.cern.ch/images/OPEN-PHO-ACCEL-2013-056-1. [Online; accessed November 16, 2021].

[12] Ehud Duchovni. The ATLAS Detector and Subsystems, 2010. URL https://www.researchgate.net/figure/The-ATLAS-detector-and-subsystems_fig1_226619417. [Online; accessed November 16, 2021].

[13] CERN. Computer generated image of the ATLAS inner detector, 2008. URL https://cds.cern.ch/images/CERN-GE-0803014-01. [Online; accessed November 16, 2021].

[14] CERN. The Inner Detector of ATLAS Detector, 2021. URL https://atlas.cern/discover/detector/inner-detector. [Online; accessed November 16, 2021].

[15] CERN. Computer Generated image of the ATLAS calorimeter, 2008. URL https://cds.cern.ch/images/CERN-GE-0803015-01. [Online; accessed November 16, 2021].

[16] CERN. Calorimeter, 2021. URL https://atlas.cern/discover/detector/calorimeter. [Online; accessed November 16, 2021].

[17] CERN. Computer generated image of the ATLAS Muons subsystem, 2008. URL https://cds.cern.ch/images/CERN-GE-0803017-01. [Online; accessed November 16, 2021].

[18] CERN. Muon spectrometer, 2021. URL https://atlas.cern/discover/detector/muon-spectrometer. [Online; accessed November 22, 2021].

[19] CERN. Atlas detector magnet system, 2021. URL https://cds.cern.ch/images/ATLAS-PHOTO-2021-029-1. [Online; accessed November 16, 2021].

[20] Joao Pequenao CERN. Computer generated image of the atlas inner detector, 2008. URL https://cds.cern.ch/record/1096081. [Online; accessed January 10, 2022].

[21] CERN. Trigger and Data Acquisition, 2022. URL https://atlas.cern/Discover/Detector/Trigger-DAQ. [Online; accessed March 16, 2022].

[22] Operation of the ATLAS trigger system in Run 2t. Technical report, CERN, 2020. URL https://iopscience.iop.org/article/10.1088/1748-0221/15/10/P10004.

[23] CERN. Data and simulated data, 2020. URL http://opendata.atlas.cern/release/2020/documentation/visualization/data-and-simulated-data_13TeV.html.

[24] Jamil Moughal. Which Machine Learning algorithm to use? 2018. doi: https://mjamilmoughal.medium.com/which-machine-learning-algorithm-to-use-bd9f7dc479c4.

[25] IBM Cloud Education. Machine learning, 2020. URL https://www.ibm.com/cloud/learn/machine-learning.

[26] Pekka Parviainen. Ensemble methods, August 2020. URL https://mitt.uib.no/courses/24958/files/2750006/download?wrap=1. [Lecture slides; University of Bergen].

[27] Eilam Gross Ofer Vitells Glen Cowan, Kyle Cranmer. Asymptotic formulae for likelihood-based tests of new physics? 2010. doi: https://link.springer.com/content/pdf/10.1140/epjc/s10052-011-1554-0.pdf.

[28] CERN. Magnet system, 2021. URL https://atlas.cern/discover/detector/magnet-system. [Online; accessed November 22, 2021].

[29] IBM Cloud Education. Recurrent neural networks, 2020. URL https://www.ibm.com/cloud/learn/recurrent-neural-networks#toc-what-are-r-btVB33l5.

[30] Pekka Parviainen. Decision tree, August 2020. URL https://mitt.uib.no/courses/24958/files/2685799/download?wrap=1. [Lecture slides; University of Bergen].

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.