Implementation of Machine Learning Algorithms to Form Di-Muons from Off-Shell Parent Particles

Mehdi Rahmani & Stephen D. Butalla

July 14, 2021

Florida Institute of Technology

Introduction

- A Fermionic dark matter model with four muons final state is considered [1, 2]
- Signal events in this model have a topology in which the final state muons come from off-shell parent particles
- Defining a signal region in this analysis requires forming correct di-muons, i.e., paring muons that come from the same parent particle

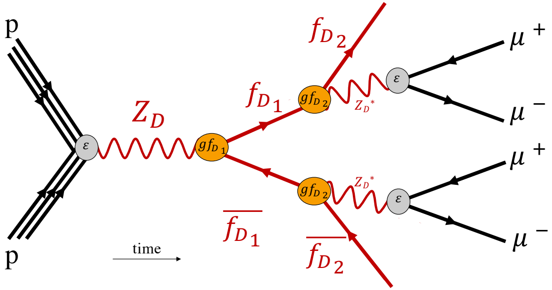


Figure 1: Feynman diagram for the fermionic dark matter model, referred to as the f_D model. The dark Z boson (Z_D) provides a **vector portal** to the dark sector through kinetic mixing. Dark Fermions, f_{D_1}'s, then decay to di-muons through off-shell Z_D particles and another stable dark fermion, f_{D_2}, that escapes detection.

Introduction

- To pair the muons into di-muons coming from same kind of parent particles, one would reconstruct the invariant mass for all possible permutations
- Out of all combinations of muons to form di-muons, the arrangement with closest invariant mass is selected
- This method is *not* applicable to our case, because the invariant dimuon masses, products of *off-shell* Z_D's, cover a whole range of values
- That means one could not rely on requiring two masses to be the same to assign muons correctly to dimuons
- Alternatively, several Machine Learning (ML) algorithms are employed to achieve this goal
- The Reconstructed (Reco) data is labeled through matching with Generated (Gen) level muons
- Having the reco muons labeled enables us to use supervised ML algorithms to solve this problem
- Observables associated with the di-muons are studied based on the network predictions
- Signal region for the analysis is defined and the shape of the signal determined

Data Generation and Processing

- Event data are generated using `MadGraph5_aMC@NLO_2_6_5` [3]
- Hadronization and showering are simulated using `Pythia8.230` [4]
- Event reconstruction is performed under `CMSSW_10_2_18` [5]
- The Monte-Carlo (MC) samples consist of a scan over the Z_D mass ($125 < m_{Z_D} < 200$ GeV), as well a scan over f_{D_1} mass ($5 < m_{f_{D_1}} < 55$ GeV), for each given Z_D mass. The f_{D_2} mass for the time being is kept at 2 GeV across all samples
- In total 80 samples are produced, each containing 10^4 events
- For the purpose of maximizing the number of events to be fed to the network, the selection is kept limited to requiring a pseudo-rapidity of $|\eta| < 2.4$ for each of the muons
- About 4000 events per sample survive selection after cuts are applied

Matching and Labeling

- Data are preprocessed by applying cuts to remove events with failed reconstruction and charge misidentification of final-state muons

- Algorithm is developed to match the reco muons to their gen muon counterparts as a means to label the reco muons with their parent particles:

- The matching algorithm proceeds as follows:
  1. Angular separation (ΔR) between the reconstructed gen muons and all 4 reco muons is calculated
  2. First gen muon is sampled randomly and *matched* with the reco muon with the smallest ΔR
  3. The other same-charge muon is paired with the remaining reco muon of same charge
  4. One oppositely charged gen muon is selected and matched via the minimum ΔR with the 2 remaining reco muons
  5. Remaining reco muon is paired with gen muon
- After this matching procedure, the permutation (order of muons) is defined as the *correct permutation*, and assigned a label 1

Matching and Labeling

- Requiring that each di-muon comprises 2 oppositely charged muons, we define 2 *wrong permutations* for each event
- Both incorrectly generated permutations are labeled 0
- To each permutation (one correct and two wrong), we associate the event features and feed them to the network

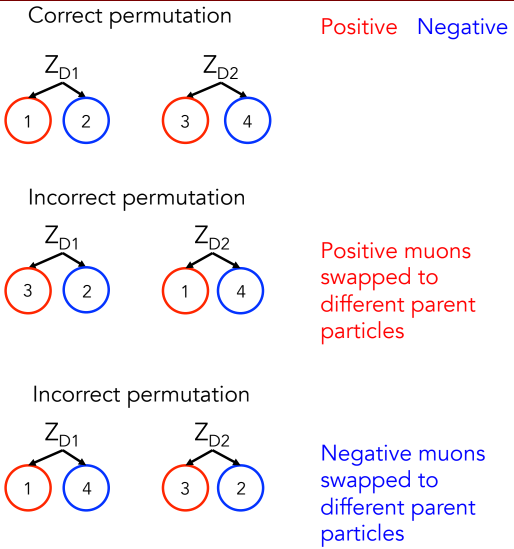


Figure 2: The correct permutation and the other 2 allowed (but incorrect) permutations.

Matched ΔR

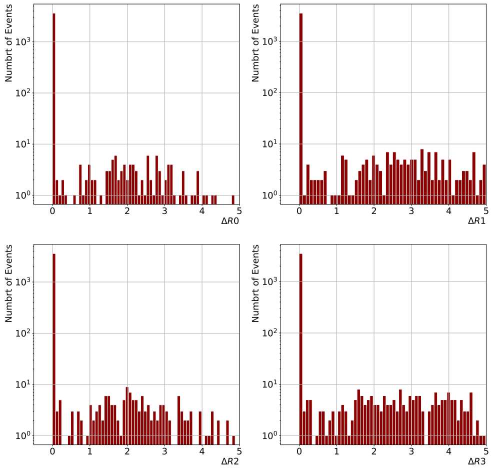


Figure 3: The ΔR between the matched *reco muons* to *gen muons* based on the developed algorithm. Most matches exhibit very low ΔR (good match). There are also events were the matches are not near perfect ($\Delta R > 0.005$; bad match). These bad events are cut.

Matched ΔR

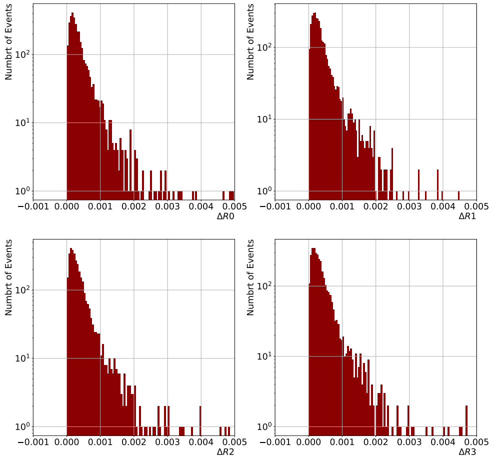


Figure 4: The ΔR between the matched *reco muons* and *gen muons* after that events with $\Delta R > 0.005$ are cut out. These events can be trusted to give us the correct parent particles for the reco muons.

Deep Neural Networks

- **Fully connected, deep, feed-Forward Neural Networks** of various topologies created using Keras [6] with a Tensorflow backend
- Data was partitioned and randomly shuffled into a 50/20/30 train/validation/test split
- Hyperparameters were manually tuned
- Final architecture selected had 9 hidden layers:

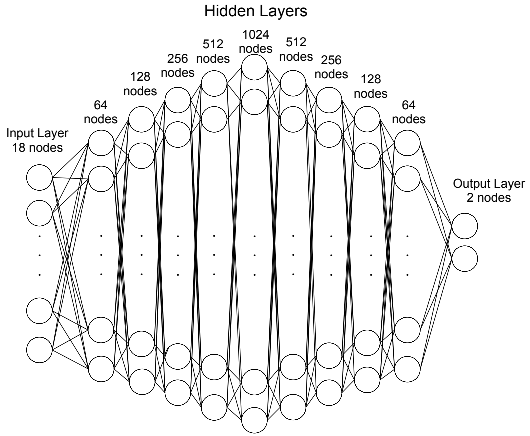


Figure 5: Final neural network architecture selected.

Final Neural Network Architecture Selected and Result

- After many iterations of hyperparameter tuning, the best classification accuracy was achieved with:
  - Features scaled to be in $[0, 1]$
  - Batch size: 32
  - Binary Cross-entropy loss function
  - Adagrad [7] (Adaptive gradient algorithm) with initial learning rate $\alpha = 10^{-3}$ (with α reduction when validation loss plateaued for more than 10 epochs)
  - 9 hidden layers with 64-1024 nodes
  - Elastic net regularization (L1 penalty: 10^{-2}, L2 penalty $= 10^{-4}$)
  - PReLU activation function
  - 30% Dropout
  - He [8] normal distribution of node weights
  - Biases initialized to 0
- 67.2% test accuracy

Neural Network Results

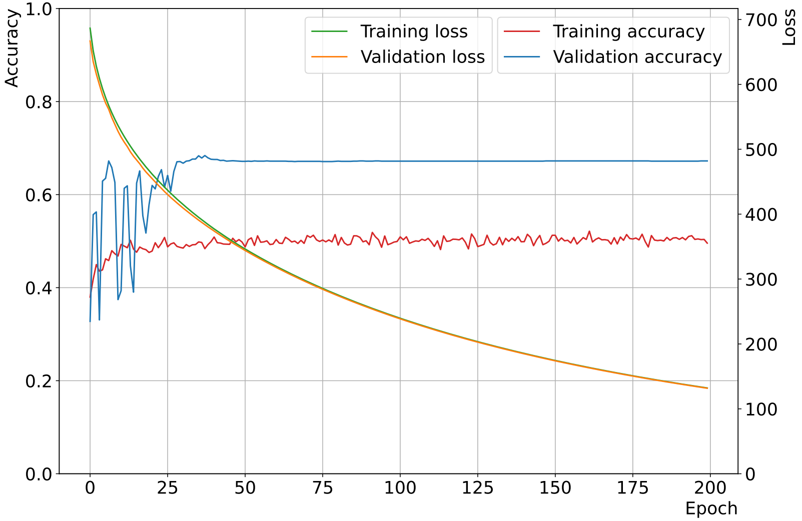


Figure 6: Loss and accuracy as a function of epoch.

Neural Network Accuracies

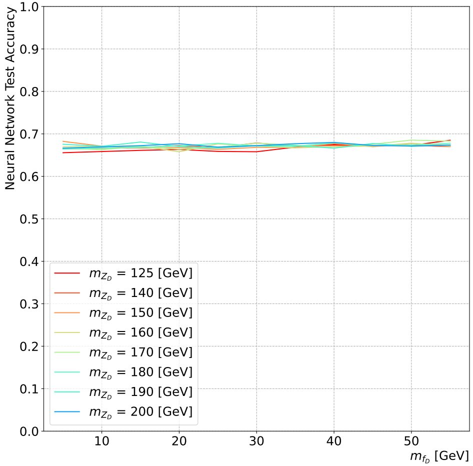
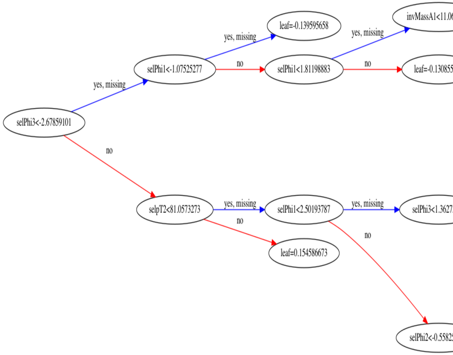


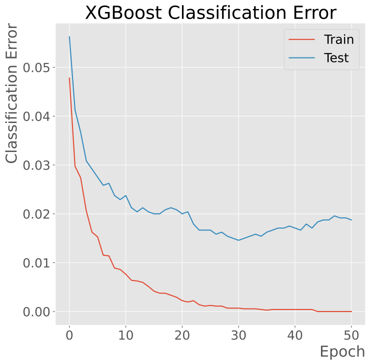
Figure 7: Accuracy of correct predictions for the neural network trained on MC simulated samples with $m_{Z_D} = 200$ GeV and $m_{f_{D1}} = 30$ GeV. Note that the performance is independent of m_{Z_D} and $m_{f_{D1}}$.

XGBoost

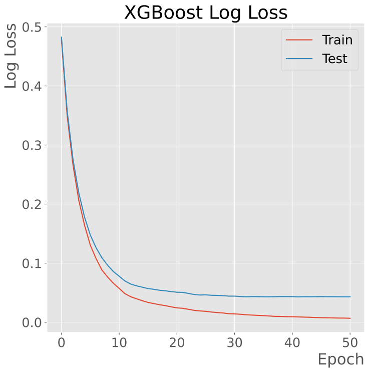
- In an attempt to improve the accuracy of the results, we turn to boosting methods; gradiant boosting in particular
- Previously gradient boosting algorithms only focused on improving impurity
- **eXtreme Gradient Boosting** (`XGBoost`) applies a variety of regularization techniques to avoid over-fitting [9]
- Bayesian hyperparameter tuning did not show significant improvement in the scores of `XGBoost` compared to default hyperparameters
- Default hyperparameters are used for `XGBoost` here: $\gamma = 0$, $\eta = 0.3$, *max depth* $= 6$, ...



XGBoost Score scans



(a) XGBoost Classification error



(b) XGBoost Log-Loss scores

Figure 8: XGBoost performance with the network trained with $m_{Z_D} = 200$, $m_{f_{D_1}} = 30$ GeV.

Di-muon Invariant Mass

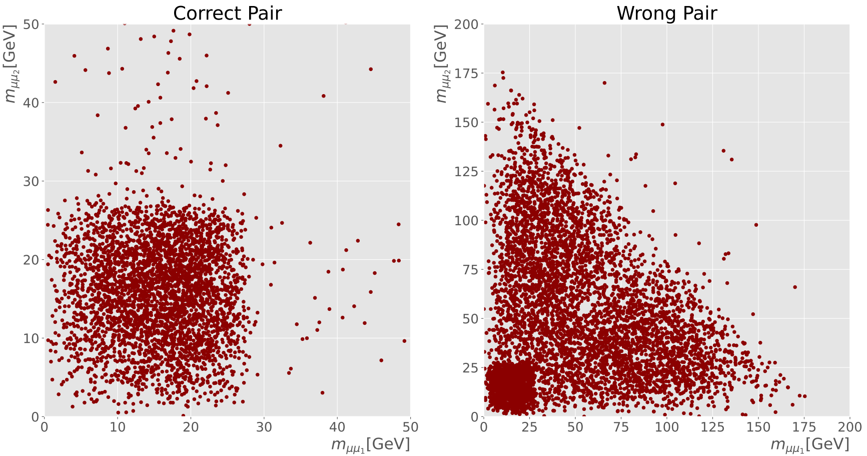


Figure 9: 2D invariant mass distributions for correctly and incorrectly paired di-muons.

Di-muon Invariant Mass

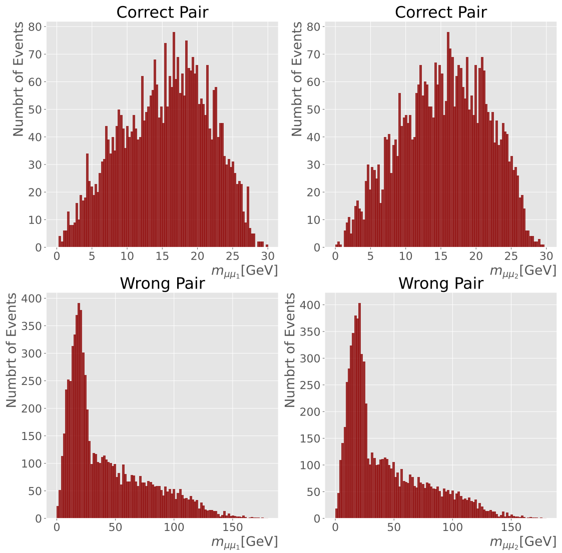


Figure 10: 1D invariant mass distributions for correctly and incorrectly paired di-muons.

XGBoost **Accuracy**

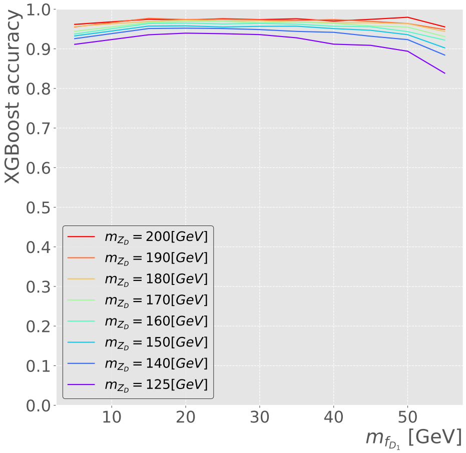


Figure 11: The Network is trained on the MC sample with the highest masses, i.e. $m_{Z_D} = 200$, $m_{f_{D_1}} = 55$ GeV, then the achieved model is saved and tested on the rest of the samples.

ROC Curves

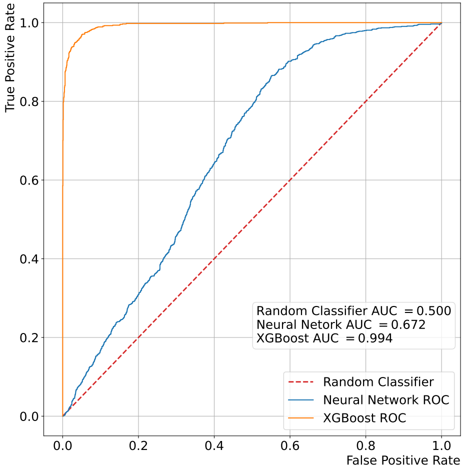


Figure 12: Accuracy of correct predictions for the neural network trained on MC simulated data with $m_{Z_D} = 200$ GeV and $m_{f_{D1}} = 30$ GeV.

Summary and Conclusions

- MC samples generated for a dark fermionic model
- Simulated events used to form di-muons that share the same parent particle for each event
- Supervised ML algorithms, such as `XGBoost` and deep, feed-forward, fully connected neural networks were examined
- `XGBoost` shows the highest classification accuracy at $\sim 95\%$
- Neural networks with hyperparameter tuning is the second best performing network with an accuracy of $\sim 67\%$
- These models were tested on samples with different masses to ensure the model is not over fitting on a particular mass
- A model-independent behavior (across samples with difference masses) is observed for both network scores

References

[1] D. Curtin, R. Essig, S. Gori, & J. Shelton, "Illuminating dark photons with high-energy colliders," *Journal of High Energy Physics*, **2015**(157), 2015, doi:10.1007/JHEP02(2015)157.

[2] J. Wells, "How to find a hidden world at the Large Hadron Collider," 2008, arXiv:0803.1243.

[3] MG/ME Development Team, `MadGraph5_aMC@NLO` (v.2_6_5), `http://madgraph.phys.ucl.ac.be/`.

[4] Pythia Development Team, `Pythia` (v.8.230), `https://pythia.org/`.

[5] CMSSW Development Team, `CMSSW` (v.10.2.18), `https://cms-sw.github.io/`.

[6] The Keras Development Team, `Keras` (v.2.3.0), `https://keras.io/`.

[7] J. Duchi, E. Hazan, Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research*, **12**(2121-2159), 2011, `https://jmlr.org/papers/v12/duchi11a.html`.

[8] K. He, X. Zhang, S. Ren, & J. Sun, "Deep Residual Learning for Image Recognition," 2015, arXiv:1512.03385.

[9] Chen, Tianqi and He et. all, "XGBoost: eXtreme Gradient Boosting", *R package version 0.4-2*, 2015

Backup

An Introduction to Neural Networks

- Artificial Neural Network (ANN) is a collection of mathematical functions (**neurons**) that receive N inputs
- Neurons modeled on the neuron of a human brain
  - Take input "signals" along with weights and biases, pass them through an activation function
- Input data called **features**; in our case, kinematic variables and higher-level, computed observables (e.g., invariant mass)
- **Activation function**: The mathematical function of the neuron; e.g.,
  - Parametric Rectified Linear Unit (PReLU): $f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$
- Data passes through the network (**feed-forward**), and at the end, the loss function value is computed
- **Loss function**: the function you are trying to minimize (training a neural network is just one big optimization problem); e.g.,
  - Binary cross-entropy (log-loss):

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -N^{-1} \sum_{i=1}^{N} y_i \cdot \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

- The data is **back-propagated** through the network, and the data are fed forward for another pass
- After each pass, the weights and biases input to the neurons are tuned (this is the learning part)

An Introduction to Neural Networks

- **Optimizer**: the algorithm used for optimizing the loss function, e.g., Adagrad:

$$w_{s+1,i} = w_{s+1,i} - \frac{\alpha}{\sqrt{\Omega_{s,ii} + \epsilon}} \nabla L(\boldsymbol{\theta})$$

  where w are the weights, α is the learning rate, Ω is a diagonal matrix with entries representing the summed, squared gradients, and ϵ used to prevent division by zero
- Neural network is run for a predetermined number of **epochs** (iterations)
- Many **trainable parameters** (all of the weights, biases)
- Many **hyperparameters**: learning rate α, number of nodes/hidden layers, regularization methods to reduce overfitting (L^1, L^2), dropout, etc.
- Number of nodes in output layer represent the categories (categorization problem) or coefficients (regression problem)
- Generally, data is split into **train/validation/test** datasets so network performance can be evaluated after training (the validation/test sets are data the network hasn't "seen")

An Introduction to Neural Networks — Preventing Overfitting

- Overfitting occurs when the model “learns” the data too well
- Can use regularization techniques, such as the L^1 and L^2 vector norms, which penalize (constrains) weights that explode, helping network stability
  - Additional terms representing the norm of the weights added to the loss function, i.e.,

$$\hat{L} = L(\mathrm{y}, \hat{\mathrm{y}}) + \lambda ||\mathrm{w}||_p$$

  - Norm:

$$||\mathrm{w}||_p = \left(\sum_i^N |w_i|^p \right)^{1/p},$$

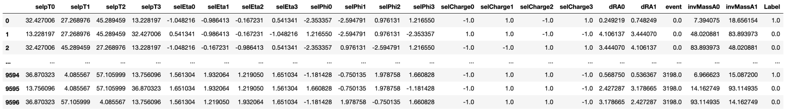
  where $p = 1, 2, \cdots, \infty$

- Dropout randomly switches a fraction of the nodes in a layer to help “shift” the learning responsibility to other nodes

An Introduction to `XGBoost`

- **Decision Tree:** a predictive model to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). Classification trees is our interest.
- A single tree work great with the data used to create them, but they are not flexible when it comes to classifying new data.
- One decision tree is prone to over-fitting.
- To reduce the risk of overfitting, models that combine many decision trees are preferred.
- **Gradient Boosting**: Combining a learning algorithm in series to achieve a strong learner from many sequentially connected weak learners. At each iteration of the gradient boosting procedure, we train a base estimator (single tree) to predict the *gradient descent* step. Saving these base estimators in memory is what enables us to output predictions for any future sample.
- **Regularization**: Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be.
- **eXtreme Gradient Boosting**: Previously gradient boosting models only focused on improving impurity (a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset.). "`XGBoost`" applies a variety of regularization techniques to avoid over-fitting.

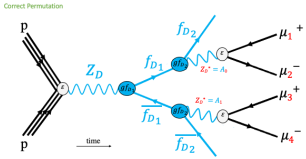
Data frame



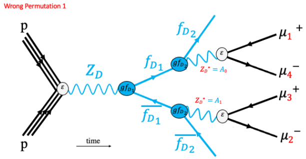
	selpT0	selpT1	selpT2	selpT3	selEta0	selEta1	selEta2	selEta3	selPhi0	selPhi1	selPhi2	selPhi3	selCharge0	selCharge1	selCharge2	selCharge3	dRA0	dRA1	event	invMassA0	invMassA1	Label
0	32.427006	27.268976	45.289459	13.228197	-1.048216	-0.986413	-0.167231	0.541341	-2.353357	-2.594791	0.976131	1.216550	-1.0	1.0	-1.0	1.0	0.249219	0.748249	0.0	7.394075	18.656154	1.0
1	13.228197	27.268976	45.289459	32.427006	0.541341	-0.986413	-0.167231	-1.048216	1.216550	-2.594791	0.976131	-2.353357	1.0	1.0	-1.0	-1.0	4.106137	3.444070	0.0	48.020881	83.893973	0.0
2	32.427006	45.289459	27.268976	13.228197	-1.048216	-0.167231	-0.986413	0.541341	-2.353357	0.976131	-2.594791	1.216550	-1.0	-1.0	1.0	1.0	3.444070	4.106137	0.0	83.893973	48.020881	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9594	36.870323	4.085567	57.105999	13.756096	1.561304	1.932064	1.219050	1.651034	-1.181428	-0.750135	1.978758	1.660828	-1.0	1.0	1.0	-1.0	0.568750	0.536367	3198.0	6.966623	15.087200	1.0
9595	13.756096	4.085567	57.105999	36.870323	1.651034	1.932064	1.219050	1.561304	1.660828	-0.750135	1.978758	-1.181428	-1.0	1.0	1.0	-1.0	2.427287	3.178665	3198.0	14.162749	93.114935	0.0
9596	36.870323	57.105999	4.085567	13.756096	1.561304	1.219050	1.932064	1.651034	-1.181428	1.978758	-0.750135	1.660828	-1.0	1.0	1.0	-1.0	3.178665	2.427287	3198.0	93.114935	14.162749	0.0

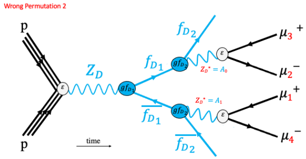
Figure 13: The data frame structure. All features do a 3 way permutations.

Feynman Diagrams



Correct permutation





Wrong permutation - scenario one

Wrong permutation - scenario two

XGBoost Classification Report

Table 1: Classification report

Training classification	precision	recall	f1- score	support
0	1.00	1.00	1.00	4785
1	1.00	1.00	1.00	2412
accuracy			1.00	7197
macro ave	1.00	1.00	1.00	7197
weighted ave	1.00	1.00	1.00	7197

Test classification	precision	recall	f1- score	support
0	0.99	0.98	0.99	16
1	0.96	0.98	0.97	787
accuracy			0.98	2400
macro ave	0.98	0.98	0.98	2400
weighted ave	0.98	0.98	0.98	2400

XGBoost Performance i

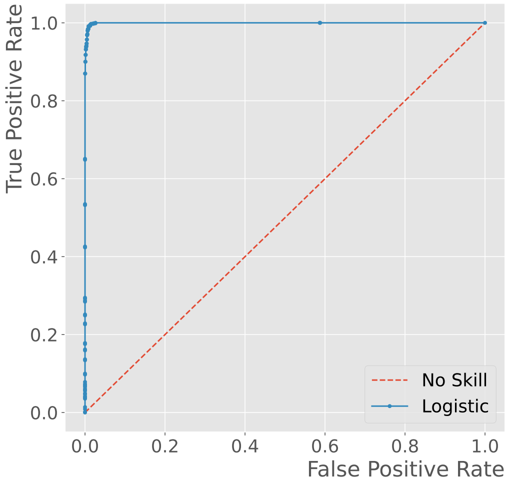


Figure 14: XGBoost Receiver operating characteristic, **ROC** plot.No Skill: ROC AUC=0.500, Logistic: ROC AUC=0.999

`XGBoost` Hyper Parameters i

- η **[default=0.3, alias: learning_rate]:** Step size shrinkage used in update to prevents over-fitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative. Range: [0,1]
- γ **[default=0, alias: min_split_loss]:** Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative the algorithm will be. range: $[0,\infty]$

XGBoost Hyper Parameters ii

- **max_depth [default=6]:** Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. 0 is only accepted in loss guided growing policy when tree_method is set as hist or gpu_hist and it indicates no limit on depth. Beware that XGBoost aggressively consumes memory when training a deep tree. Range: $[0,\infty]$ (0 is only accepted in loss guided growing policy when tree_method is set as hist or gpu_hist)
- **min_child_weight [default=1]:** Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger min_child_weight is, the more conservative the algorithm will be. Range: $[0,\infty]$

XGBoost Hyper Parameters iii

- **max_delta_step [default=0]:** Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced. Set it to value of 1-10 might help control the update. Range: $[0,\infty]$
- **subsample [default=1]:** Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration. Range: (0,1]
- **sampling_method [default= uniform]:** The method to use to sample the training instances. uniform: each training instance has an equal probability of being selected. Typically set subsample ¿= 0.5 for good results.

XGBoost Hyper Parameters iv

- **gradient_based:** the selection probability for each training instance is proportional to the regularized absolute value of gradients subsample may be set to as low as 0.1 without loss of model accuracy. Note that this sampling method is only supported when tree_method is set to gpu_hist; other tree methods only support uniform sampling.

- **colsample_bytree, colsample_bylevel, colsample_bynode [default=1]:** This is a family of parameters for subsampling of columns. All colsample_by* parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled.
  - **colsample_bytree** is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.
  - **colsample_bylevel** is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree.

XGBoost Hyper Parameters v

- **colsample_bynode** is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.
- **colsample_by*** parameters work cumulatively. For instance, the combination 'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5 with 64 features will leave 8 features to choose from at each split.
- On Python interface, when using hist, gpu_hist or exact tree method, one can set the **feature_weights** for DMatrix to define the probability of each feature being selected when using column sampling. There's a similar parameter for fit method in sklearn interface.

- λ **[default=1, alias: reg_λ]:** L2 regularization term on weights. Increasing this value will make model more conservative.
- α **[default=0, alias: reg_α]:** L1 regularization term on weights. Increasing this value will make model more conservative.