Summer Student Report

July - September 2023

Supervised by *Pieter Everaerts*

Machine learning for displaced vertex classification

Ruben Bekaert
ruben.lukas.bekaert@alumni.cern
Vrije Universiteit Brussel
(+32) 487 71 30 72

Abstract

Displaced Vertices are an important signature for new physics searches at the Large Hadron Collider. They point to the existence of long-lived massive particles, which are predicted by many Beyond the Standard Model theories. In this project, we investigate the use of machine learning techniques to classify these displaced vertices in an R-parity violating supersymmetry (SUSY) model, as a simple cut and count can have low signal efficiency. We apply Boosted Decision Trees (BDTs) to secondary vertex information, and Graph Neural Networks (GNN) to fully connected graphs of tracks in the detector. While the BDTs are able to provide an immediate improvement in signal efficiency over the cut and count, the GNNs warrant further investigation.

Contents

1 Introduction **2**
1.1 Searching for Displaced Vertices 2
1.2 Stop Quark Decay 2
1.3 Cut and Count 2
2 Boosted Decision Trees **3**
2.1 Ensemble Learning 3
2.2 Gradient Boosting 3
2.3 Implementation 4
2.4 Results and Discussion 4
3 Graph Neural Networks **6**
3.1 Graph Convolution 7
3.2 The Attention Mechanism 7
3.3 Implementation 7
3.4 Results and Discussion 8
4 Conclusions **9**
Bibliography **10**

1 Introduction

1.1 Searching for Displaced Vertices

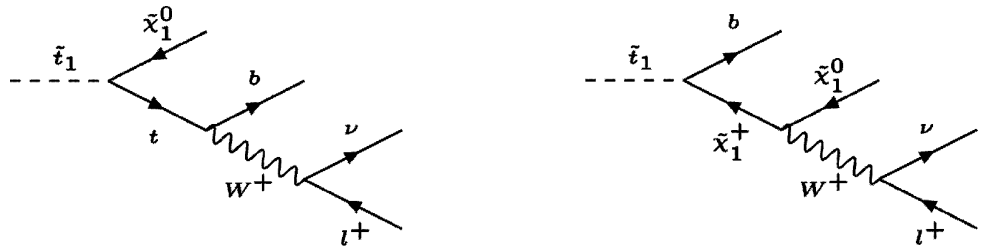
A feature of many supersymmetric (SUSY) extensions of the standard model is the occurrence of new long-lived massive particles. These give rise to features such as disappearing tracks, or secondary vertices displaced from the collision point, which provides a good target for searches of SUSY processes.

The challenge in finding these is that SUSY cross-sections for the LHC at the expected masses are very small [1]. Since these are often many orders of magnitude smaller than the background processes, it is imperative to find a way to cut out as much of the background as possible.

Evolving machine learning techniques have allowed for vast improvements in classification techniques. The aim of this project is to explore their suitability for displaced vertex classification.

1.2 Stop Quark Decay

The stop is the supersymmetric partner of the top quark. It is typically the lightest squark, and is therefore the most likely to be produced at the LHC. In this analysis, the focus lies on a R-parity violating SUSY model in which two stops are produced, each decaying to a lepton and a b-jet or d-jet. The stops are long-lived, so they travel some distance before decaying, resulting in a pair of displaced vertices. This signature is used as a proxy in the search for these long-lived stops.



(a) Top channel of stop decay.

(b) Chargino channel of stop decay.

Figure 1: Decay channels of the stop squark in 'natural SUSY' [2].

Background processes that can mimic this signature include top quark pair production, or W boson production in association with jets, as well as a range of QCD processes.

1.3 Cut and Count

A common hand-crafted classification algorithm is the simple cut and count. This technique involves looking at the difference in distributions of discriminant variables for signal and background. One can then decide a cutoff for when a data point is regarded as signal. This defines a rudimentary decision boundary [3].

The main advantage of a cut and count is that it is fully controlled. Most machine learning models with a high degree of generalisability quickly lose interpretability. This is a major trade-off, as we want to be able to understand the model in order to trust it.

For the purposes of the search conducted here, the cut and count is only able to preserve roughly 70% of the signal at a background rejection of 99%. This performance varies on the assumed mass and lifetime of the stop, with signal efficiencies for low mass, short lifetime stops suffering the most. These have signal efficiencies as low as 30%, leading to the conclusion that a cut and count is not sufficient for the search at these masses and lifetimes. Clearly, there is still a significant amount of signal left on the table, warranting the use of machine learning.

2 Boosted Decision Trees

2.1 Ensemble Learning

Ensemble learning is a collection of machine learning techniques where a large amount of weak learners together can form a strong learner [4]. These weak learners can be any model, so long as they perform slightly better than random chance. Most often, they will be small decision trees with some limited depth. The idea is that each of these weak learners will be able to capture a small part of the data, so that when combined together, they can capture the entire data set.

The simplest form of ensemble learning is bagging, where each weak learner is trained on a random subset of the data. Their predictions are then averaged together to form the final prediction. If the weak learners are decision trees, this is called a random forest.

However, there are smarter ways to combine weak learners in ways that ensure they are not all learning the same thing. One such method is boosting, where each weak learner is trained on the same data set, but the trees are trained in sequence in such a way that each tree is trained to correct the mistakes of the previous ones.

2.2 Gradient Boosting

The principle of gradient boosting can be applied to learn an approximation to any function $F(x)$, using any differentiable loss function. The loss function L is a function of the estimated values $\hat{y}$ and the true values y, that is minimized when the estimator is close to the true value. A typical example is the squared error loss function, which is defined as:

$$L(y, \hat{y}) = \sum_i \left(y_i - \hat{y}_i\right)^2 \tag{1}$$

The function we wish to learn is approximated by a series of M weak learners $f_{i(x)}$.

$$\hat{F}(x) = \sum_{i=1}^{M} f_{i(x)} \tag{2}$$

These weak learners are trained in sequence. We start with function $F_0(x) = \vec{0}$. We then want to train a weak learner $f_1(x)$ to minimize the loss function on the data set. First, the negative gradient of the loss function is computed with respect to current estimator function.

$$r_0^{(i)} = -\frac{\partial L\big(y^{(i)}, \hat{y}^{(i)}\big)}{\partial \hat{y}^{(i)}}\bigg|_{\hat{y}^{(i)}=F_0(x)} \quad , \tag{3}$$

where $\bigcirc^{(i)}$ denotes the i-th element of the dataset. The new weak learner f_1 is then trained to predict r_0 from x. Since we want to descend the gradient of the loss function, we know that the new estimator function $F_1(x)$ should be equal to $F_0(x) + \gamma_1 f_1(x)$. The value of γ_1 is found by doing a line search to minimize the loss function.

$$\gamma_1 = \text{argmin}_\gamma \sum_i L\big(y^{(i)}, F_0(x) + \gamma f_1(x)\big) \quad . \tag{4}$$

Doing this, our new estimator $F_1(x)$ should be better than the previous estimator. This process is then repeated for M iterations, where each new weak learner is trained to predict residuals of the previous estimator. The final estimator is then the sum of all the weak learners.

To use this as a classifier instead of for regression, the output of $F(x)$ has the dimensionality of the number of classes, and the class with the highest output value is chosen.

2.3 Implementation

There are a number of popular libraries implementing gradient boosted decision trees (BDTs), such as XGBoost, LightGBM, CatBoost, and scikit-learn. For the purposes of this project, both XGBoost and scikit-learn were tested.

While it would be possible to do some event classification on the basis of event-level information, we are looking for model that can specifically identify displaced vertices. This means we'll be training on vertex-level information instead, including the following features:

- the number of tracks in the vertex,
- the distance between the vertex and the beamspot in the transverse plane,
- the squared sum of the transverse momenta of all tracks in the vertex,
- the cosine of the angle between the momentum of the vertex tracks and the distance from the beamspot to the vertex.

The BDTs are trained on datasets of stop decays for stop masses ranging from 200 GeV to 1800 GeV, and lifetimes between 0.1 mm 30 mm. The background samples mentioned in Section 1.2 are also included. Due to the unbalanced amount of training data in the classes, each event type is weighted according to the expected yield (from the cross section, and the effectiveness of previous selection cuts) and the inverse of the number of training events. Total signal yield is set to be equal to background yield for the training. For XGBoost, it is also important to scale the weights such that they are all roughly of order 1.

The dataset is split into a training set and a test set. The training set is used to train the BDT, and the test set is used to evaluate the performance of the BDT. The test set is not used in any way during the training process, so it allows us to get an unbiased estimate of the performance of the BDT, and gauge the level of overfitting to training data.

2.4 Results and Discussion

While scikit-learn was able to yield great results faster, XGBoost proved to be more configurable, yielding slightly better results after careful tweaking [5]. The best results were obtained using binary logistic regression with a learning rate of 0.1, histogram binning, a maximum depth of 3, and about 500 estimators. More than this only leads to overfitting. scikit-learn's results were only marginally worse, and it makes for an easy plug-and-play solution to evaluate performance expected from BDTs.

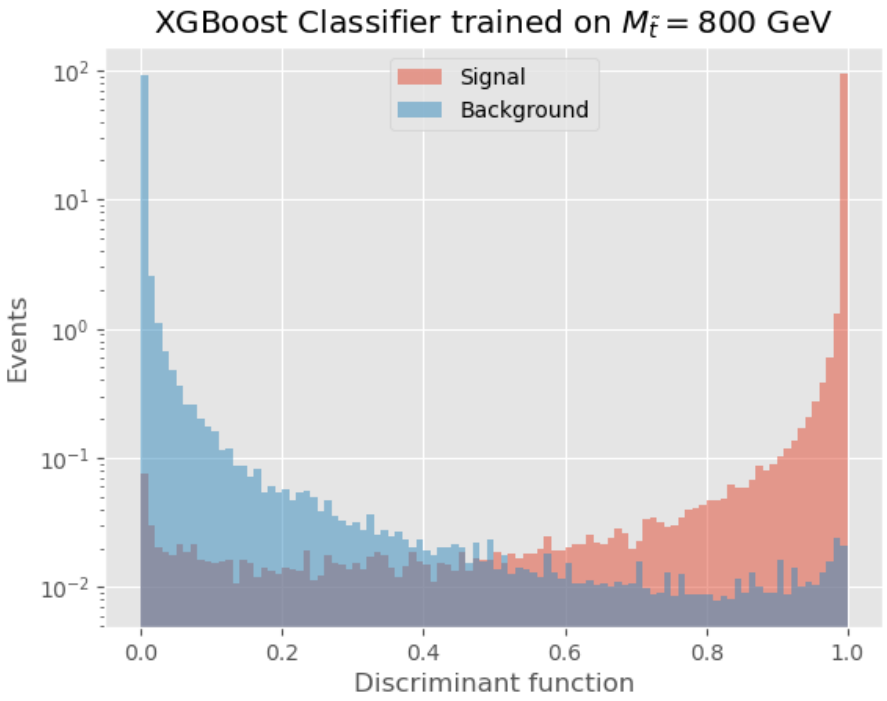


Figure 2: BDT discriminant output for signal and background.

Figure 2 shows the separation the BDT is able to achieve between signal and background. Both profiles clearly peak on their own end of the output spectrum, and are suppressed by about 4 orders of magnitude on the other end.

The performance comparison with the cut and count is shown in Figure 3. The BDT is able to achieve a much better signal efficiency for the same background efficiency. Even for the lowest stop mass and lifetime, which is the worst-case scenario for the BDT, it still outperforms the highest efficiency cut and count, which only ever reaches about 90% efficiency on the heaviest and longest living stops.

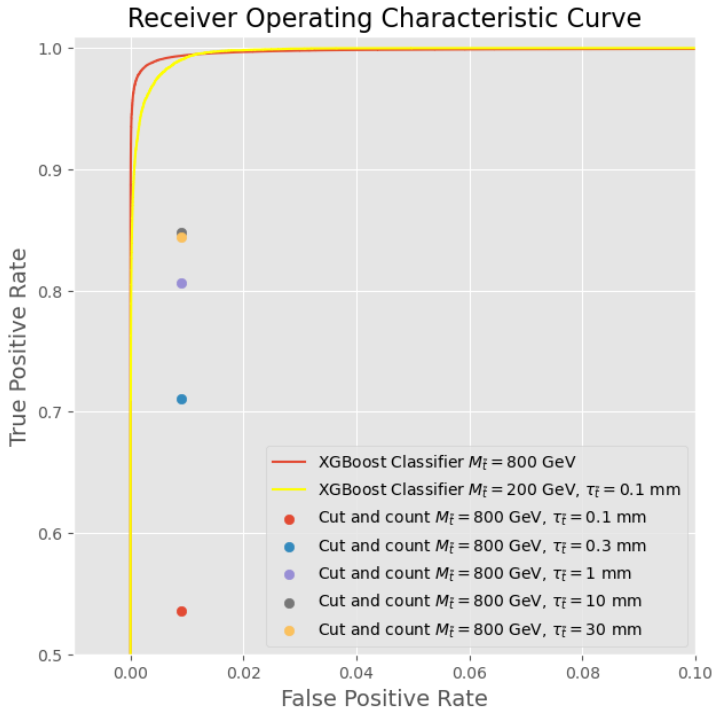


Figure 3: ROC curve for the BDT compared to cut and count.

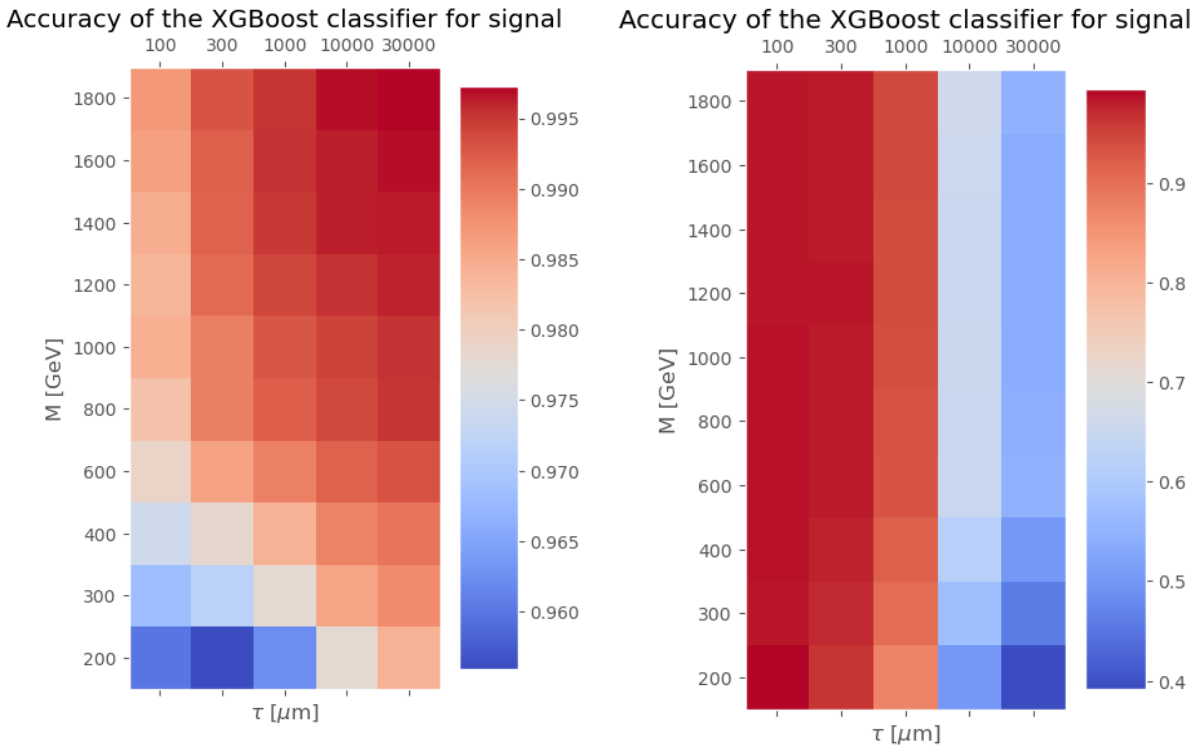


Figure 4: Efficiency of the BDT on each of the signal samples. Left: trained on $M_{\tilde{t}} = 800$ GeV and the full lifetime range, Right: trained on $M_{\tilde{t}} = 200$ GeV and $\tau_{\tilde{t}} = 0.1$ mm to $\tau_{\tilde{t}} = 1$ mm.

A BDT trained on some specific mass or lifetime can be applied on signals with different properties, and still distinguish it from background. In fact, as seen from Figure 4, it gets even better at distinguishing higher-mass, higher-lifetime stops than what it was trained on. The worst results are obtained on the lightest, shortest-lived stops, but even there, a BDT trained on $M_{\tilde{t}} = 800$ GeV is able to achieve the same efficiency as a BDT trained on just that single signal sample. This shows the BDT is learning some generalisable properties of displaced vertices, which can be applied to any signal sample.

This is however not universally true. On the right of Figure 4, the signal efficiencies for a BDT trained on only the lightest, shortest-lived stop sample is shown. This BDT is able to achieve good efficiencies on short-lived stops of any mass, but fails to recognise longer-lived stops. This demonstrates that it was unable to learn properties of displaced vertices that generalize to those signal samples.

3 Graph Neural Networks

During the vertex reconstruction step of CMS's event reconstruction, a lot of secondary vertices will be missed. Therefore, if it is possible to do event classification without having to rely on the reconstructed vertices, this may help in getting a better signal-to-noise ratio at the end.

From just singular tracks, it is difficult to get a good image of the event topology. However, it should be possible when looking at how the tracks relate to each other. This is where graph neural networks come in. They are able to take into account the relations between the tracks, and therefore should be able to get a better image of the event topology.

There have been previous successes in doing similar things with raw detector hits in simplified environments [6], and CMS makes extensive use of the ParticleNet Graph Neural Network (GNN) for jet tagging [7].

3.1 Graph Convolution

The graph convolution is the main building block of the graph neural network. It is a generalization of the convolutional layer in a Convolutional Neural Network (CNN) [8].

Every graph consists of a set of nodes and (directed) edges. Both nodes and edges can have any number of features. The graph convolution is a function that takes as input a graph, and outputs a graph with the same number of nodes, but with updated node features.

Information in the graph convolution of an arbitrary graph isn't as neatly structured as in the grid convolution of the CNN. Information must be propagated from one node to another through the edges. This is done through 'message passing'. At each of the nodes, a message is computed from its features. This message is passed over all the nodes to its neighbors. At the neighbors, the messages are aggregated together with the node's local features. This aggregation needs to be a permutation invariant function (as there is no inherent order in the graph), and must have the same output dimensionality irrespective of how many messages the node receives. This is usually an average of the messages, but could be max or some other function.

The simplest graph convolution is the Graph Convolutional Network (GCN). Its messages are the node features themselves, and the aggregation is a normalised average of the passed messages. The GCN layer at node v defined as follows:

$$\boldsymbol{x}_v^{(\ell+1)} = \boldsymbol{W}^{(\ell+1)} \sum_{w \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{n_v n_w}} \boldsymbol{x}_w^{(\ell)} \quad , \tag{5}$$

where $\boldsymbol{W}$ is a matrix of learnable weights, $\mathcal{N}(v)$ is the set of neighbors of node v, n_v is the number of neighbors of node v, and $\boldsymbol{x}_v^{(\ell)}$ is the feature vector of node v at layer ℓ. After each of these convolution layers, a non-linear activation function is applied to the node features. This is usually a ReLU or sigmoid.

GNNs can be used for node-level predictions, for graph-level predictions, or for edge-level predictions. In this case, we are interested in graph-level predictions. This means that at the end of the GNN, the node features are averaged out to get a single feature vector for the graph. This feature vector is then fed into a fully connected neural network to get the final output.

Since there can only be a finite number of layers, the information from the nodes far away from the current node may not be able to reach it. In k layers, information can travel at most k nodes along the graph. However, due to the nature of the graph convolution, information is averaged out every layer. Therefore, all features in the graph tend to smooth out after only a couple layers. This is not ideal, as it limits the spread of information, and also makes deep learning with GNNs incredibly hard [9].

3.2 The Attention Mechanism

One method that has been used to improve the spread of information is the Graph Attention Network (GAT) layer. This is a graph convolution that is inspired by the attention mechanism in the Transformer architecture [10]. It allows the network to focus on certain parts of the input, and ignore others by computing an attention score for each of the messages based on the features and connecting edge. The message aggregation is then done by a weighted average of the messages using the attention scores. This way, more important information can be propagated further, and nodes can focus on the information that is most relevant to them [8].

3.3 Implementation

From track reconstruction, we have a list of seedtracks, with features such as:

- a reference position on the track,

- the direction of the track,
- the momentum of the track,
- number of detector hits.

From these seedtracks, we can construct a graph. The nodes of the graph are the seedtracks. However, there is no way to reasonably connect tracks without some kind of vertexing step, which is precisely what we want to avoid. Therefore, we will create a fully connected graph by connecting all tracks to all other tracks. This is not ideal, as it means computation scales quadratically with the number of tracks. As is evident in Figure 5, this is a lot of edges, and may be computationally expensive for large amounts of tracks.

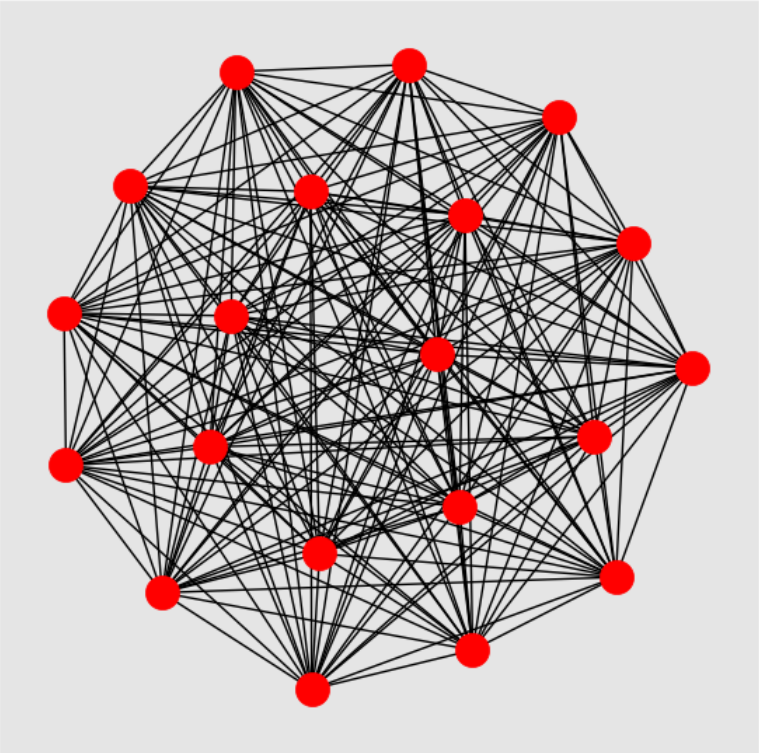


Figure 5: A fully connected graph of seedtracks

With a simple GraphConv model, the network is completely unable to learn anything. However, by using the Graph Attention Network (GAT) instead, the network is able to look through the noise of the fully connected graph, and learn the correct topology. For this, we use the GAT implementation from the PyTorch Geometric library [11].

To help focus the GAT, we add edge features by computing the distance between the seedtrack reference positions. If the tracks pass close to eachother, they are more likely to be from the same vertex.

The network is built from three layers of Graph Attention Network, with 32 hidden channels each. During training, we use a dropout rate of 0.3 to prevent overfitting and produce a more robust network. At the end of the network, the node features are averaged, passed through a single linear layer and fed through the softmax function to get the final output.

3.4 Results and Discussion

After only a single epoch of training (15000 events), the Graph Attention Network is able to distinguish the signal graphs from the background much better than random guessing. This indicates that the network is able to correctly learn which tracks might be from the same vertex, and whether or not

said vertex is displaced. Results improve slightly in roughly the following 5 epochs, after which further training quickly loses efficacy. At 99% background rejection, the resulting signal efficiency is 95%.

Comparing the GNN's performance to the BDTs, we find that the BDT appears to be both more performant and robust. As can be seen in Figure 6, both background and signal have significant peaks on the wrong side of the GNN's discriminant. It is however impossible to directly compare the performance of the GNN to the BDTs. This is because the BDTs are applied after vertexing and some preselection cuts. This means that even with a lower signal efficiency, the GNN could potentially still keep more signal overall, as a lot of secondary vertices are missed in the vertexing step.

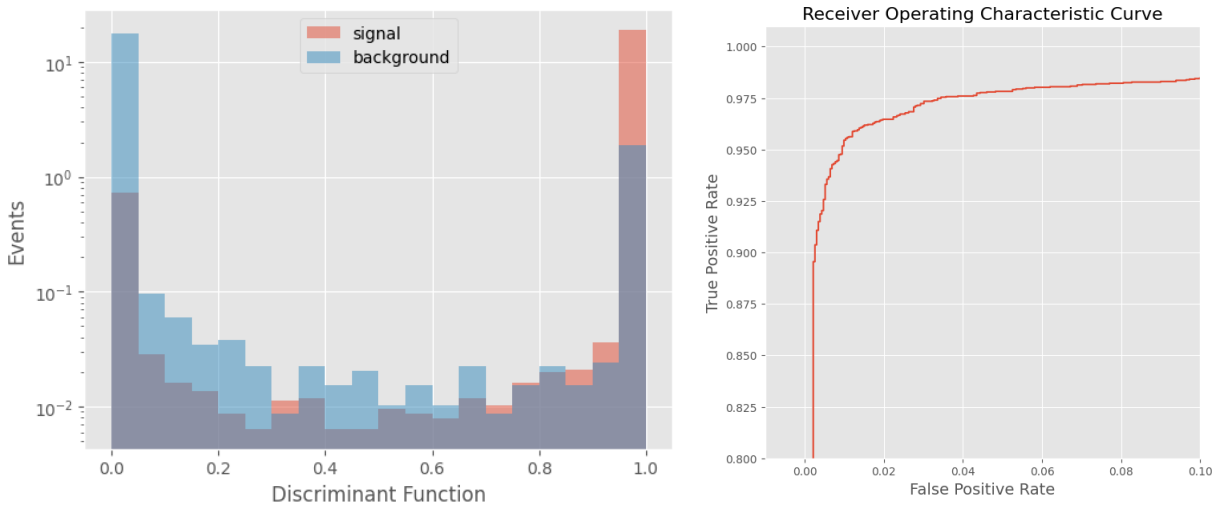


Figure 6: Left: Discriminant output for the signal and background graphs. Right: ROC curve for the GNN.

As it is, the GNN is far from ready for use in a real analysis. If it is possible to connect the seedtracks in a better way, the performance could still be further improved. So far, we have not been able to crack this problem.

From the ROC curve, it is evident that the GNN suffers from low number of simulated events. Training on a larger dataset of simulated events could lead to significant improvements in performance, as well as a more robust network.

4 Conclusions

Machine learning techniques are a useful tool in classification in searches for processes beyond the standard model. We have shown that BDTs are excellent at classifying displaced vertices to find long-lived massive particles, beating the more traditional cut and count.

To leverage the graph-structured data of hits in a particle detector, Graph Neural Networks are a promising avenue to explore. They seem to be able to learn more of the underlying structure of the event, but they require a lot more development before they could be used in this analysis. Especially the graph-building step warrants reconsidering, and is currently very computationally expensive.

Bibliography

[1] A. Mann, "Stop-antistop production cross sections in proton-proton collisions at 13 tev," 2021. [Online]. Available: https://twiki.cern.ch/twiki/bin/view/LHCPhysics/SUSYCrossSections13TeVstopsbottom (LHCPhysics Web)

[2] L. Wu, and H. Zhou, "Polarization of top and chargino from stop decay in natural SUSY," *Phys. Lett. B*, vol. 794, pp. 96–102, Jul. 2019, doi: 10.1016/j.physletb.2019.05.033. [Online]. Available: https://doi.org/10.1016/j.physletb.2019.05.033

[3] B. Coleppa, G. B. Krishna, A. Sarkar, and S. Shil, "Optimizing the cut and count method in phenomenological studies," arXiv, 2023. [Online]. Available: https://arxiv.org/abs/2305.10915

[4] A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras, And TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, Second edition, Beijing [China] ; Sebastopol, CA: O'Reilly Media, Inc, 2019.

[5] xgboost developers, "Xgboost: a scalable tree boosting system," 2021. [Online]. Available: https://xgboost.readthedocs.io/en/latest/ (XGBoost Documentation)

[6] K. Albertsson, and F. Meloni, "Ctd2020: displaced event classification using graph networks," 2020, doi: 10.5281/ZENODO.4034420. [Online]. Available: https://zenodo.org/record/4034420

[7] H. Qu, and L. Gouskos, "Jet tagging via particle clouds," *Physical Rev. D*, vol. 101, no. 5, Mar. 2020, doi: 10.1103/physrevd.101.056019. [Online]. Available: https://doi.org/10.1103/physrevd.101.056019

[8] A. Daigavane, B. Ravindran, and G. Aggarwal, "Understanding convolutions on graphs," *Distill*, vol. 6, no. 8, Aug. 2021, doi: 10.23915/distill.00032. [Online]. Available: https://doi.org/10.23915/distill.00032

[9] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, vol. 6, no. 8, Aug. 2021, doi: 10.23915/distill.00033. [Online]. Available: https://doi.org/10.23915/distill.00033

[10] A. Vaswani, N. Shazeer, et al., "Attention is all you need," arXiv, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[11] P. G. developers, "Pytorch geometric: geometric deep learning extension library for pytorch," 2021. [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html (PyTorch Geometric Documentation)