Collision Simulation Reweighting with Machine Learning in Nuclear and Hadron Physics

Alex Hodgson, 9 April 2020

E-mail: `alexander.whodgson@gmail.com`

Abstract.
Simulated collider experiment data has many uses in nuclear and hadron physics, though the accuracy to which we can recreate nature in these simulations is limited. To make the simulated data as close to real data as possible, and therefore more useful, an adjustment can be applied through reweighting. A machine learning algorithm can be trained to identify which simulated collisions appear the most like real data then weight these more heavily so that the simulated data better represents real data. A reweighter based on gradient boosted regression trees applied with a folding algorithm was found to provide good results when compared with adaboost and multi layer perceptron neural network based reweighters. Two datasets with different distributions were generated, one to represent real data and one the simulated data to evaluate the reweighting methods' performance. When comparing the two distributions before and after reweighting with the folding gradient boosted regression trees algorithm, the Kolmogorov-Smirnov statistic was reduced from 0.031708 ± 0.001249 to 0.0039860 ± 0.0008855 and the Kullback-Leibler divergence was reduced from 0.026274 ± 0.0005559 to 0.0019997 ± 0.0001332. This good performance continued when the complexity of the data was increased, suggesting the reweighting algorithm can be extended for use in multiple scenarios.

1. Introduction

Monte Carlo (MC) simulations play an important role in nuclear and hadron physics, allowing data sets that would be difficult or time consuming to measure physically to be generated relatively quickly and easily. These data sets have uses such as calibration of detectors and event selection algorithms along with predicting background noise and what signals would be expected to be seen if some physical process is occurring in a collision [1]. These simulations will differ slightly from real measurements of the same events that are being simulated for a variety of reasons that are not easily corrected for, including available processing power limiting the depth of the physics being considered or aspects of a detector's performance that haven't been accurately modeled [2]. So that the generated data is as close to real data as possible to ensure accuracy in the above uses, once all reasonable corrections have been applied, the data can be reweighted with some machine learning algorithm so the distribution of MC data more accurately represents real results. Though in practice the training of the reweighter would be conducted with some simulated data and some real historic data from a detector experiment [2], the aim of this project was to find a method of reweighting that can be used on any two multivariate distributions so the source of the each data set is of little significance, as long as the two distributions are different. Both data sets used in this project are therefore simulated, though in different processes to ensure differences between the distributions. The data set used to represent the real data that simulated data is being reweighted to better represent is then called the 'real' data in this report. Section 2 provides a general overview of machine learning and classification algorithms, to provide useful background information. Section 3 describes how the data sets were generated, giving consideration to the kinematics of a particle collision. In Section 4 further detail is provided on the machine learning algorithms that were applied and Section 5 explains the most successful reweighting algorithm used in this project.

2. Machine Learning Theory

In general, Machine Learning (ML) is the process of giving a data set to some algorithm to build a model that can give predictions about the nature of some new data on which the model is applied. The algorithms used are often computationally intensive, and, in the case of deep neural networks, make decisions where the motivation behind them is very difficult for humans to understand, which is why these calculations are done computationally [3]. The methods discussed in this report are all based on *classification algorithms*, a form of supervised learning that works by taking some labeled data, where each data point is labeled with which class it belongs to (in this case, if the data is 'real' or simulation data). The algorithm finds the correlations between data points with the same label and uses these to build a model that can predict the class of new unlabelled data that is passed to it.

This data will contain information on a number of *features*, which are the properties or variables that have been measured for each collision in the dataset, for each point there is a *feature vector*, which is a vector containing that point's value for each feature. Here there are either 6 or 9 features given to the algorithms for each event, dependent on the type of collision modeled. These are the energy, momentum and polar angle of each new particle in either a 2 or 3 body final state. The algorithm will be given some training data with examples of both 'real' and simulated collisions and then attempt to find the optimal way to classify the examples based on their features so that as many examples as possible are correctly labeled. So that these algorithms can be used for reweighting, a confidence score for the classification of each new example is required, this is given as a probability of the example coming from the 'real' data set. This score of how 'real' the point looks is then used to weight how much that point contributes to the histogram of the simulated data, if done accurately the weighted histograms of simulated data should then look the same as the histograms of the 'real' data.

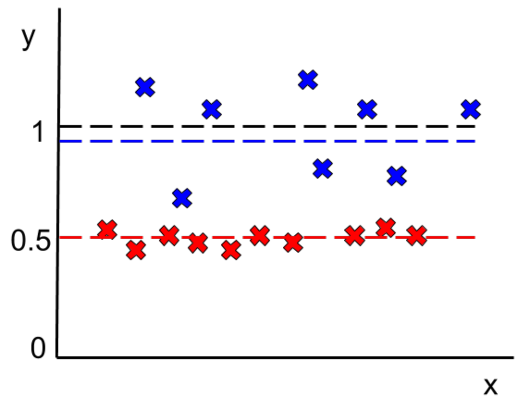

Figure 1. Using feature x to predict y where the true value of y is shown by the black dashed line. The model giving predictions in red has low variance but high bias and the model giving blue predictions low bias but high variance.

When choosing an algorithm and it's configuration, a balance must be found between the minimisation two sources of error. *Bias* error is caused by a model's failure to learn the trends in the data due to underfitting, causing a model to be consistent in it's incorrect predictions. *Variance* error is caused by the model learning the training data too well and including random fluctuations in the training data in it's predictions, a case of the model overfitting. An overfitted model cannot be generalised as the it incorporates random noise from the training set in it's model and would have large errors if applied to unseen data. Simple algorithms such as linear regressions are more vulnerable to high bias as they cannot represent the full complexity of the data, while complex algorithms such as decision trees can give models with high variance if precautions are not taken to prevent this overfitting. A simple example of these errors is visualised in Figure 1.[3]

Each algorithm used for this project has one of two validation methods, which are used to test the accuracy of the model built by using that model to predict classes for data it wasn't trained on and doesn't know the labels for. The first method is to perform a simple *test/train split* which divides the data into two sections, the first part is used to build the model and the second part is used to check if collisions from the same data set that were not used to train the model can have their classes predicted accurately. The other method is *k fold cross validation* where the data is split into some integer k parts (folds) of equal size, k reweighters are then produced with each one being trained on k - 1 folds. For each reweighter the fold not being used to train the model is then used to validate the predictions of the model as with the previous split method [4]. A voting/averaging system can be used to take into account the predictions of every model, which can be beneficial if the individual models have a high variance error.

2.1. Evaluation of reweighting methods

To measure similarity between the 'real' and simulated distributions two separate metrics were used so that different quantitative results could be compared between reweighting methods. The two sample Kolmogorov–Smirnov (KS) test gives a likeliness that two data sets were drawn from the same distribution by producing an empirical distribution function (EDF) $F_n(x)$ for each data set as shown in (1)

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{I}\{X_i \leq x\} \tag{1}$$

Where $X_1...,X_n$ are the n independent and identically distributed ordered observations from the distribution being tested, and $\mathbf{I}\{X_i \leq x\}$ is the indicator function equal to 1 if $X_i \leq x$ and 0 otherwise [5]. The KS Statistic is then the largest distance between these functions:

$$D_{n,m} = sup_x(F_n(x) - G_m(x)) \tag{2}$$

Where $G_m(x)$ is the EDF of the second distribution and *sup* is the supremum function [6]. This distance $D_{n,m}$ is the Kolmogorov-Smirnov statistic and the smaller it is, the more similar the distributions. The KS test was chosen for use here as it is a *non parametric* test, so can be used to find distances between distributions of any shape, whereas similar alternative tests require one data set to be distributed in a known form (eg normal or exponential) [7].
The Kullback–Leibler (KL) divergence was also used to evaluate the similarity of two distributions, and by extension the effectiveness of reweighting algorithms. The KL divergence of distribution Q from distribution P can be considered a measure of information lost if Q is used to represent P. This divergence is [8]:

$$D_{KL}(P||Q) \equiv \int_{-\infty}^{+\infty} p(x) \log \frac{p(x)}{q(x)} dx \tag{3}$$

Where p(x) and q(x) are the probability density functions of P and Q respectively. In this case P and Q were approximated as discrete probability density functions using the bins the data had been collected into, care was taken to ensure enough bins were used to so the value found for D_{Kl} was a good approximation. The smaller D_{Kl} is, the more similar the distributions are and a divergence of zero would mean the distributions are identical [9]. Like the KS test, KL divergence can also be used with distributions of any shape, though care must be taken with the bin approach used here. If a bin of P contains events but the bin of Q it is compared with does not, $D_{Kl} = \infty$ and the two distributions will be found to be absolutely different. It should also be noted that $D_{Kl}(P||Q) \neq D_{Kl}(Q||P)$ so the comparison between original and reweighted distributions must be consistently done as the divergence of the reweighted from the target for meaningful comparison between methods.

3. Data generation

Here both the simulated and 'real' data sets for the ML algorithms to reweight are generated in the same manner then adjusted separately to create two different distributions to test an algorithm's ability to reweight one set of data to be closer to another. Different adjustments and physical processes are investigated to ensure the algorithms can be applied generally and that the results could be extended to data from detector experiments.

3.1. Collision simulation

The collisions being simulated are those that would occur in a fixed target experiment, where a beam of accelerated particles is incident on some stationary target, such as liquid hydrogen. The collisions were modeled simply considering conservation of energy and momentum in the centre of mass (COM) frame of the beam and target particles, where the initial particles are destroyed and produce two daughter particles. Psuedo random numbers are generated with the Mersenne Twister for the polar and azimuthal (θ, ϕ) angles which one new particle travels along, with the other moving in the opposite direction to conserve the zero net momentum of the particles in this frame. This creates an even spread of paths in all directions in the COM frame before the particles are then boosted into the lab frame to simulate the paths that would be seen by an observer (Figure 2). Four-vectors are used to describe the particles during the simulation of the collision to simplify calculations for the mechanics of the collision and the Lorentz boosting of the particles. For the 3 body final state collisions, one of the particles formed in the initial collision decays into two smaller particles, conserving the energy and momentum of that particle. Once the four-vectors for these new particles have been calculated they are also boosted into the lab frame.

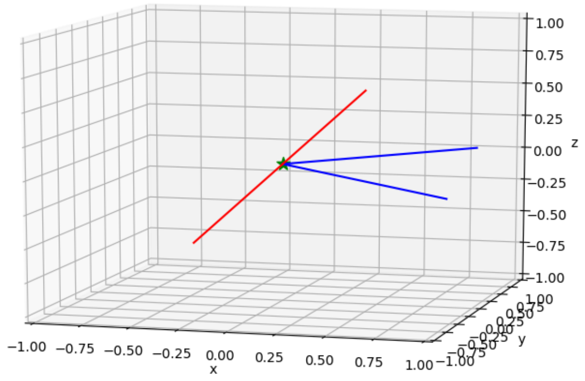

Figure 2. Paths after collision in COM frame (red) and lab frame (blue)

3.2. Detector Simulation

To create differences between the original and target data sets a number of adjustments were applied to both sets, with varied parameters between the original and target sets to change the severity and form of the functions being applied. The final distributions after these simulations were applied are shown in Figure 3.

Three aspects of a detector's performance were simulated and the effects of these were applied to the collision data already generated. To simulate the failure of the detector to count particles that traveled nearly perpendicular to the beam line (small θ), collisions that created particles with a θ value smaller than some threshold would be discarded, as most detectors have a solenoidal shape with an opening at each end.

To imitate a detector's resolution in measuring energy and momentum, a small gaussian blur was applied to each measurement of these variables. The distribution would have a mean of the value previously generated and some standard deviation given as a parameter of the current simulation, a new value for energy or momentum would then be chosen with the height of the distribution at some energy or momentum value being the probability that that value is taken.

As the probability of a detector producing a signal from a particle passing through it is likely to depend on that particle's energy, an acceptance function was also implemented. This can be set to linear, quadratic or a step function to test a variety of different cases. A line is drawn of probability of detection against energy based on the input parameters, then for each collision a psuedo random float in the range [0.0, 1.0) is generated, if the random value is less than the acceptance function at that collision's energy the event is kept, otherwise it is discarded.

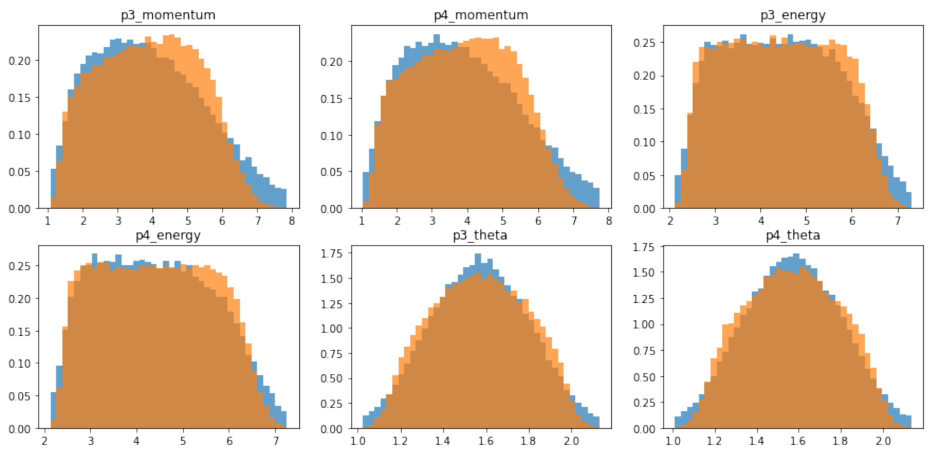

Figure 3. Two body collision with no weights applied, 'real' data in orange, simulated data in blue

4. Initial Reweighting Methods

Seven different reweighting methods, based on a number of different different classification algorithms were used to weight the simulated dataset (original) in an attempt to make it match the 'real' (target) data and their success at this was measured using the KS and KL tests. Each algorithm will be given some training data consisting of both 'real' and simulated data so it can learn what a 'real' collision looks like. The algorithm is then tested when given some new simulated data, from which it will select the points that look most like 'real' data by assigning them high weights derived from each point's probability of being 'real' as calculated by the classifier. This new simulated data then has these weights applied to each point, and if successful the weighted simulated data should have a distribution similar to the 'real data.

4.1. Naïve bin reweighting

This is a simplistic approach that bases the weights given to each collision entirely upon one feature, it can be successful in cases where all the other variables are dependent on this one feature, or if you have only one feature. Using the training data one feature of the 'real' and simulated data is binned, and the ratio of 'real' bin height to simulated bin height is used as the weighting factor for all simulated collisions that fall into each bin.
This method will give good agreement between the distributions in the feature that is reweighted, and any other features that are dependent on that feature, but is unlikely to perform well with more complex data.

4.2. Neural network classifiers

The neural neworks used in this project were all *multi layer perceptrons* (MLP), which are loosely based on the functioning of a human brain. Each perceptron (or neuron, as in a brain) is connected to a number of other perceptrons in the adjacent layers which are in turn connected to some other perceptrons. A MLP has an input layer which contains all the feature values for a sample, then some amount of hidden layers of configurable width that feed values forward sequentially, and finally an output layer that has all the output values. In this project there are either 6 or 9 input values depending on the type of collision, one output value which is the probability of a collision being 'real', and the depth of hidden layers and their width was varied to investigate the effect on model performance. A visualisation of a network with 2 hidden layers each of 8 perceptrons is shown in Figure 4.

Starting with the values of the features for the sample being classified in the first layer, each neuron feeds into every neuron in the next layer with some weight w unique to the connection. There is also a bias input (not related to bias error) unique to each perceptron, this can be modeled as an input of value 1 with the neuron's bias as that input's weight. These weighted inputs are then summed and have an activation function applied to them, the rectified linear unit function (equation 4) was used in this project as it gives good performance when increasing the depth of neural networks [10], and by extension

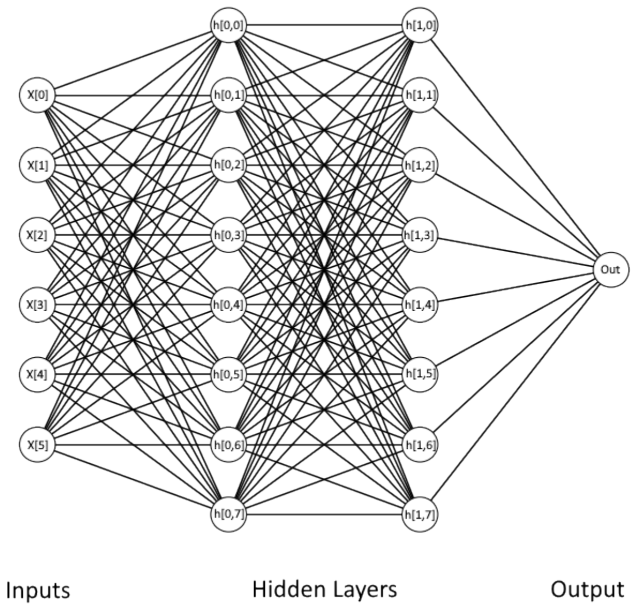

Figure 4. MLP with 2 hidden layers of width 8

the complexity of the correlations they are capable of understanding.

$$a(x) = \max(0, x) \tag{4}$$

In a multi layer perceptron such as the one shown in Figure 4, the output of neuron h_{ij}, the jth neuron in hidden layer i, with bias b_{ij} is given by equation 5 [11]. Layer i has K_i neurons and the weight on the input from neuron k in layer $i-1$ into neuron j in layer i is w^i_{kj}

$$h_{ij} = \max\left(0, b_{ij} + \sum_{k=1}^{K_{i-1}} \mathbf{h}_{i-1,k} \times \mathbf{w}^i_{kj}\right) \tag{5}$$

The learning in a MLP is done by optimisation of the weights of the neurons' connections and each neuron's bias, all the weights and biases are initialised with some random values and then optimised to minimise the cost of the network in equation 6. y_i is the label, or true value of that point's probability of being 'real', and $f(\mathbf{x}_i)$ is the network's prediction of y_i when given input $\mathbf{x}_i$. The cost of the neural network is the average of all the squared errors it makes when classifying the m points in a data set and must be minimised so the network can give accurate predictions.

$$\text{cost} = \frac{1}{m}\sum_{i=1}^{m}(y_i - f(\mathbf{x}_i))^2 \tag{6}$$

In this project two optimisation methods were tested, the adam optimiser which is a stochastic gradient descent based algorithm [12] and the l-bfgs algorithm which is a quasi-newton optimiser [13]. These both perform back-propagation when calculating the ideal values for the weights. Back-propagation calculates the partial derivative of the cost with respect to every weight and bias in the network, starting with the layer that feeds into the algorithm and working back from there [14]. This derivative is then used to find a minimum for the loss function and set the weights and biases for the most accurate network.

4.3. Ada boost classifier

Boosting algorithms, such as ada boost (Adaptive Boost), are an example of *ensemble learning.* This is when many weaker predictive models are combined in some way to make one more accurate model, ada

boost does this iteratively by combining many weak predictors which are each a little better than random guessing to create a stronger model. Here a decision stump (decision tree of depth 1) is used as the weak learner, which attempts to split the data into two sections, one 'real' and one simulated section. The split is made on the feature, and the value of that feature, for which the least amount of points are incorrectly classified. Each tree gives a point a classification probability based on the ratio of correctly to incorrectly classified training samples that were in the leaf of that point, then the results of each tree are combined for the final result.
Decision trees can be very unstable when used to classify data that is not simply separable like the data in this project, which reduces their suitability for calculating classification probabilities for individual samples [15]. A small change in the training data can lead to a completely different tree being constructed and the classification probability for one sample being very different. This instability is another motivation for using ensemble learning methods to reduce the variance of the predictions by combining the predictions of multiple trees. For training data set X with m samples in it, each with feature vector x_i and label y_i, the algorithm starts with the distribution where all samples have the same weight $1/m$ [16]

Given $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ where $\mathbf{x}_i \in X, y_i \in \{-1, 1\}$

Starting weight for each point: $D_1(i) = \dfrac{1}{m}$ for $i = i, \ldots, m$

The algorithm then makes T models sequentially, at each stage t the algorithm chooses the predictor h_t that has the lowest weighted error ϵ_t, the probability that the classifier gives the wrong label when predicting the classes of the weighted data. In this case the predictor chosen is the decision stump that classifies the training data the most accurately.

For $t = 1, \ldots, T$:
Train a weak learner using distribution D_t
Get weak hypothesis $h_t : X \to \{-1, 1\}$
Choose h_t with the lowest weighted error:

$$\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i] \tag{7}$$

Each model h_t is given a weight α_t for it's contribution to the final model based on it's error in predicting the training data then the next distribution is calculated by increasing the weights of points which were previously predicted incorrectly [17].

Choose learner weight $\alpha_t = \dfrac{1}{2} \ln \left(\dfrac{1 - \epsilon_t}{\epsilon_t} \right)$
Update the distribution, for $i = 1, \ldots, m$

$$D_{t+1}(i) = \frac{D_t(i) exp(-\alpha_t y_i h_t(x_i))}{Z_t} \tag{8}$$

Then once all T predictors have been built the final model works by combining each of their predictions for some point and considering the weight of each model:

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x) \right) \tag{9}$$

5. Gradient boosted regression tree classifier

The most success with reweighting algorithms was found using gradient boosted regression trees to classify the collisions. The accuracy was improved further when a folding method was used to reduce the model's variance.

5.1. Regression Trees

A regression tree is a form of decision tree, these trees start with all the data in one *node*, a point in the tree, called the root node and then split the data into child nodes that have some label that applies to all collisions that end up in them. A regression tree is similar to a decision tree, but the labels given to each sample are some numeric value, in this case the probability of a collision being 'real', rather than just a classification as in a decision tree. The algorithm will keep splitting nodes until it reaches some depth limit set beforehand, if splitting further would reduce the accuracy of the model or if a perfect fit is achieved. A node that is not split any further is a leaf node, when the model is applied to new data any collisions that end up here are given the probability score assigned to collisions in that leaf [18].

At each split the algorithm considers the mean squared error (MSE) of the current node and potential children of that node, to see if splitting the data will reduce the MSE. A simple example of a split (not necessarily the optimal split) is shown in Figure 5. If the algorithm was to create two new nodes, one for each side of the split, the MSE for each new node is the average of the squared distances from each point to the average value of all points in it's new node γ. The tree will consider all possible splitting points, which are the halfway points between each pair of consecutive points in each feature, then choose the one that gives the children the smallest MSE [18].

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \gamma)^2 \tag{10}$$

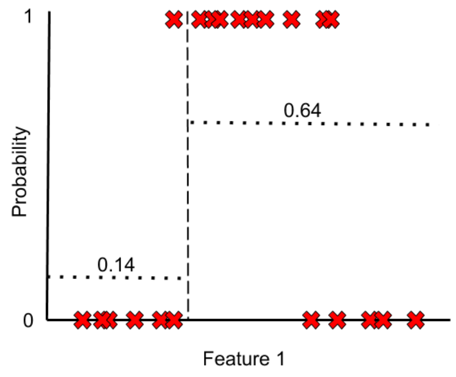

Figure 5. An example of a split in feature 1

To allow probabilities to be calculated the tree must be truncated before every training collision ends up in it's own leaf, giving all new data that ends up in that leaf a probability of either 1 or 0. Enough depth is required for the patterns to be learned, but not so much as to cause overfitting which can be an issue with poorly configured decision trees.

The first 3 levels of a regression tree trained on this collision data is shown in Figure 6. A link to a visualisation of a full sized regression tree is provided in Appendix B.

In each node in Figure 6, the feature the algorithm chose to split on and the value of the split is shown in the first line, then the mse of the data in that node, the number of collisions in the node and finally the average value of all points. This average is the probability value that would be given to any new collisions that end up in that node when the model is applied to unseen data.

5.2. Gradient Boosting

A gradient boosted regression tree is trained by taking a set of m samples, each with feature vector $\mathbf{x}_i$ and label y_i. It requires some differentiable loss function, and in this case uses a least squares method, so the distance between y_i and the classifier's prediction $F(\mathbf{x}_i)$ can be minimised. In this project y_i is a point's probability of being 'real', for the data given to the model this is either 1 or 0 as we know which set the training data came from, $F(\mathbf{x}_i)$ is then the probability the classifier gives of the point coming from the 'real' data set.

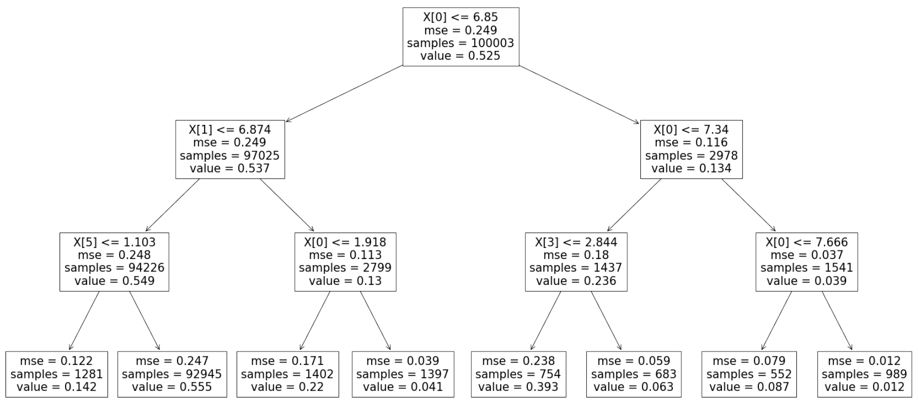

Figure 6. A regression tree of depth 3

Given Data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ and loss function: $L(y_i, F(\mathbf{x}_i)) = \sum_{i=1}^{m}(y_i - F(\mathbf{x}_i))^2$

Where $F(\mathbf{x_i})$ is the classifier's prediction for y_i

The classifier's first prediction for every sample's probability of belonging to the 'real' data is the mean value of $y_i, \ldots, y_m$ as this is the constant that gives the smallest loss.

$$\text{Initialise model with constant } F_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{m} L(y_i, \gamma)$$

For each model made starting with this original guess, the residuals (r) are calculated and then this data is used to build the next tree, which aims to be able to predict the residual for each point if all the previous models are used to classify that point.

For $t = 1, \ldots, T$:

$$\text{Calculate } r_{it} = -\left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}\right]_{F(x)=F_{t-1}(x)} \quad \text{for } i = 1, \ldots, m \tag{11}$$

Substituting the loss function gives a psuedo residual for each point:

$$r_{it} = 2(y_i - F_{t-1}(\mathbf{x}_i)) \tag{12}$$

Fit a regression tree to the values of r_{it} with leaves R_{jt} for $j = 1, \ldots, J_t$
Where J_t is the number of leaf nodes created by this iteration's tree

$$\text{For } j = 1, \ldots, J_t \text{ calculate } \gamma_{jt} = \underset{\gamma}{\operatorname{argmin}} \sum_{\mathbf{x}_i \in R_{jt}} L(y_i, F_{t-1}(\mathbf{x}_i) + \gamma) \tag{13}$$

Again substituting the loss function in here gives:

$$\gamma_{jt} = \underset{\gamma}{\operatorname{argmin}} \sum_{\mathbf{x}_i \in R_{jt}} (y_i - (F_{t-1}(\mathbf{x}_i) + \gamma)^2 \tag{14}$$

Which will give an output value γ_{jt} equal to the average value of the residuals of all points in leaf R_{jt}. The previous model's prediction for this sample is then combined with the output value for the leaf that this sample appears in to give a new more accurate prediction [19].

$$\text{Update } F_t(x) = F_{t-1}(x) + \alpha \sum_{j=1}^{J_t} \gamma_{jt} \mathbf{I}\{x \in R_{jt}\} \tag{15}$$

In equation 15, α is the *learning rate*, which is the factor that controls the rate at which the algorithm corrects it's errors which is commonly set at $\alpha = 0.1$ [20], as it was in this project, which gives a good increase in accuracy without too much of an increase the number of iterations the algorithm must go through before it gives good predictions. The algorithm is attempting to minimise the loss function $L(y_i, F(\mathbf{x}_i))$ so may overshoot the minimum if α is too large, it has been shown empirically that including this factor helps prevent overfitting and makes the model more accurate when applied to new data, compared to an $\alpha = 1$ [20].

A further increase in the ability of the model to be generalised can be achieved via *subsampling*, which is where the algorithm is only given a random fraction of the training data at each stage to build the regression tree [21]. This causes the noise in the data to change at each stage while the underlying distribution remains roughly the same, preventing overfitting and also increasing the speed of calculations as a smaller tree is being built each time. In this project a subsample fraction of 0.4 was used and found to give good results.

5.3. Folding Reweighter

Regression trees in general, including those built with gradient boosting can be vunerable to overfitting, and can be unstable when built [22]. These limitations can lead to high variance error when applying the models that have been built to new data, but a folding over reweighter can mitigate this variance error. As described in section 2, this reweighter splits the combination of 'real' and simulated data randomly into k folds, then trains k reweighters, each on $k - 1$ folds of the data. Each fold of the simulated data is then given weights by the reweighter that it was not part of the training data for. As small changes in the data set used to generate a regression tree can lead to large changes in the tree's structure [22], and each model is trained on a different part of the data, the resulting k models will all be different from each other. Every fold of the data is then reweighted by a different model so any large variances should be canceled out over the whole data set.

6. Methods

20 sets of simulated and 'real' data were generated and each had the reweighting algorithms applied to them so the reliability of the methods could also be assessed. Each set originally contained 50000 'real' and 50000 simulated collisions so there would still be enough data points after discarding some during the detector simulation. Firstly a two body final state was modeled and the parameters of the detector performance were varied between the 'real' and simulated data sets giving the distributions seen in Figure 3. Each reweighting algorithm was applied to the data and used to predict weights for each simulated point then for each reweighted distribution the average KS statistic and KL divergence were then taken across all features.
One MLP using the adam solver was made with hidden layers of size $24, 18, 12, 6$ and another was made with layers $8, 8$ to investigate the effect of network depth and width on reweighting performance. The L-BFGS network was also configured with a depth of $24, 18, 12, 6$.
For the adabost reweighter 75 decision stumps were combined to give each model. For the gradient boosting reweighter 64 regression trees were used, with a max depth of 32 and with a minimum number of samples per leaf of 200. The subsampling rate was set at 0.4 and the learning rate 0.1. The folding reweighter used the same hyperparameters as the gradient boosted rewighter and the data was split into 5 folds.

7. Results and Discussion

For each set of collisions both the KS distance and KL divergence were calculated for every variable and averaged to give one value for how close the multivariate distributions were. The KS values for the methods are shown in Figure 7 and the KL divergences in Figure 8. The green triangle marks the mean and the orange line the median. The interquartile range and total range of the results are given by the boxes and the whiskers respectively.

	KS Statistic	KL Divergence
No Weights	0.031708 ± 0.001249	0.026274 ± 0.0005559
Naive Reweighting	0.038236 ± 0.009543	0.027522 ± 0.007739
Gradient Boost	0.018903 ± 0.002756	0.0080552 ± 0.0007837
Folding GB	0.0039860 ± 0.0008855	0.0019997 ± 0.0001332
Adaboost	0.021971 ± 0.003048	0.0093990 ± 0.001035
Deep Adam NN	0.024516 ± 0.005326	0.013643 ± 0.0009311
Adam NN	0.025459 ± 0.005437	0.015587 ± 0.001349
L-BFGS NN	0.017994 ± 0.01058	0.017994 ± 0.005420

Table 1. KS and KL results from the reweighting of two body final state collisions, ± SD

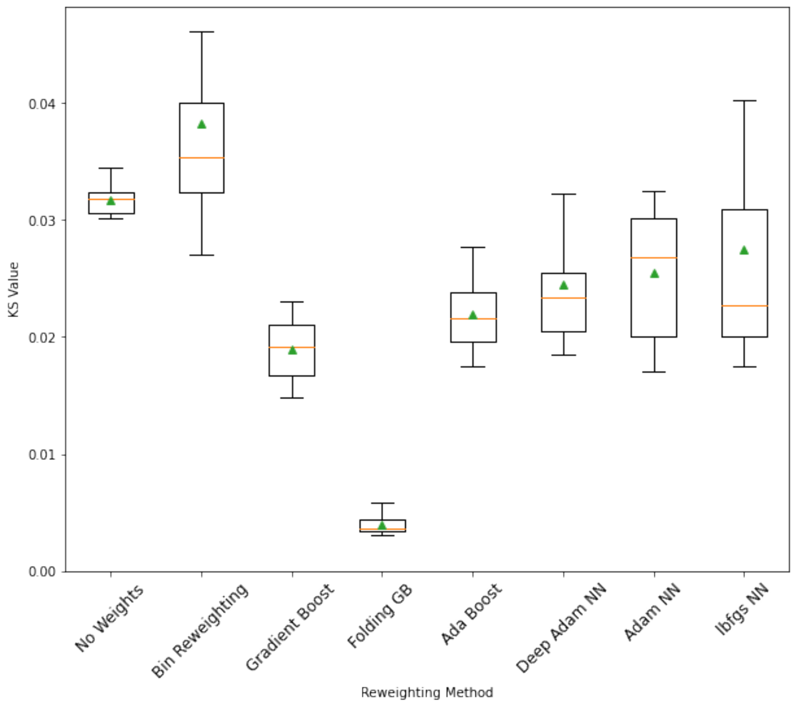

Figure 7. Average KS Statistic for each reweighting method

As can be seen from Figures 7 and 8, there is good agreement between the two metrics on the performance of the reweighting algorithm relative to each other. The naïve bin reweighter did very little to improve the simulated distribution versus no weighting, and in many cases matching up one feature caused worse agreement in the other features. In this example the momentum of particle 3 (p3) was chosen to reweight with, the agreement between the two datasets for this variable was good, but there was no improvement across the other variables. This reweighted data can be seen in Figure 9 where the target data is in orange and the reweighted data in blue.

The Neural networks performed better, with the adam solver narrowly beating the other solvers by both performance metrics, and the deeper network outperforming the shallow network. Though these methods initially looked promising by showing increased agreement with the target data, their performance could not be improved to the levels of other methods. Using a simple network caused high bias error as the network was not complex enough to understand the data, but when increasing the network

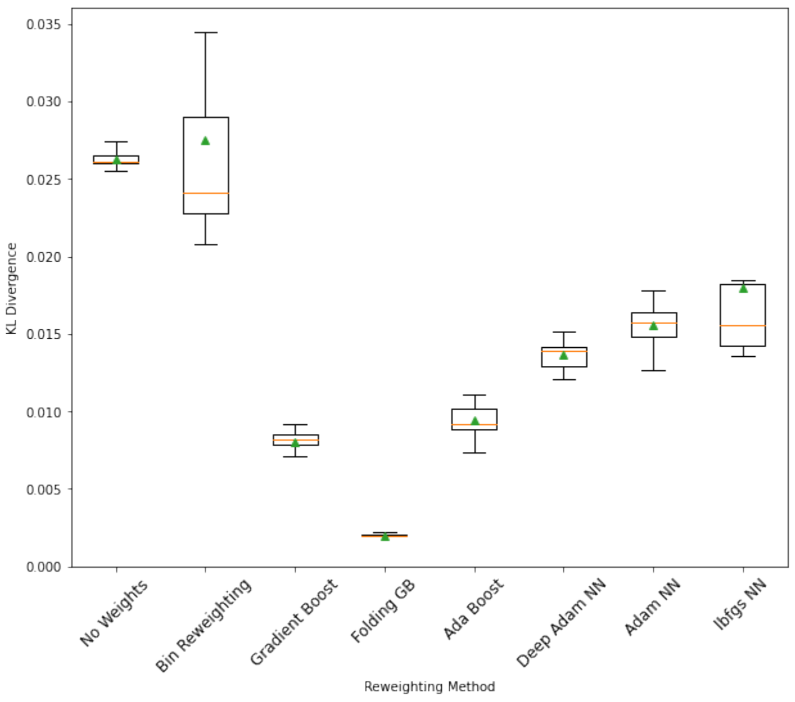

Figure 8. Average KL divergence for each reweighting method

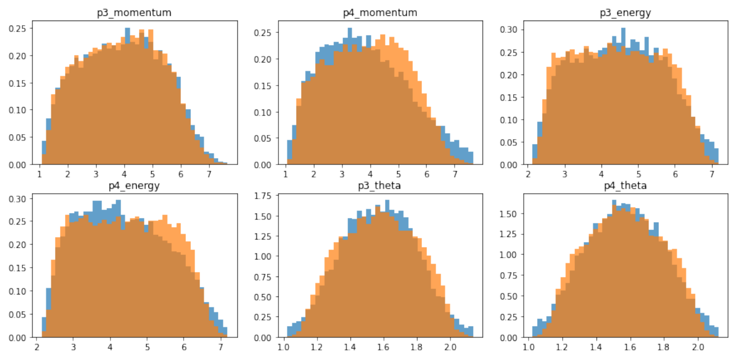

Figure 9. Simulated data reweighted naïvely based on p3 momentum

complexity a balance could not be found with these three models that allowed the data to be learned well without causing high variance errors.
The adaboost classifier gave improved performance over the neural networks but again failed to represent the learn the data fully. The bias errors can again be seen in Figure 10 where some systematic errors remain. The failiure of adaboost's simple learners to represent the data properly lead to the choice of gradient boosting as a possible reweighting algorithm, as the deeper trees it uses to classify data could allow more complex relations to be learned.

The gradient boosting classifier slightly improved the reweighting over the adaboost and neural networks, reducing the bias error as it's more complex structure could understand the data in more depth. Some variance error can be seen in Figure 11 where the heights of the simulated bins fluctuate around the heights of the 'real' bins they are targeting. This algorithm is vulnerable to overfitting, and making the trees smaller to combat this would cause the bias error to rise instead. To reduce this variance error the folding GB reweighter was used as a form of bagging, to combine the results of multiple GB reweighters and reduce variance error.

When using the same parameters for the base reweighter, the 5 fold GB reweighter was found to significantly reduce the variance error, and give very good agreement between the 'real' data and the

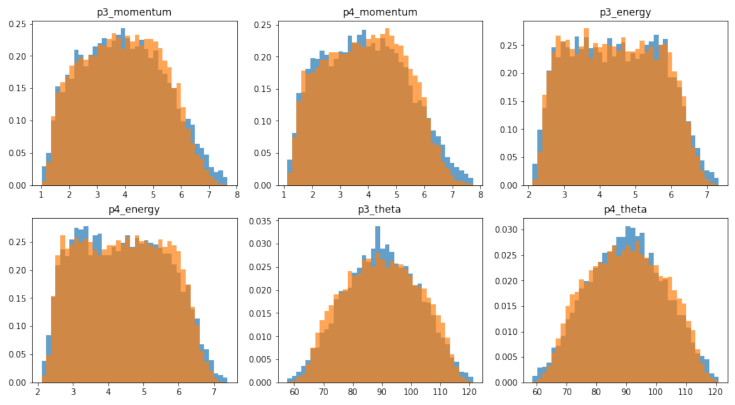

Figure 10. Simulated data reweighted using adaboost classifier

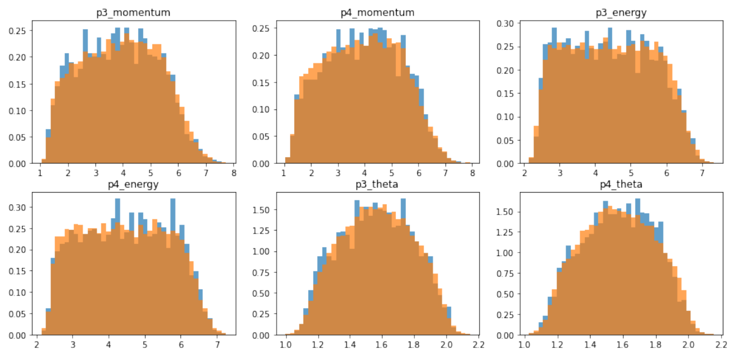

Figure 11. Simulated data reweighted using gradient boosted regression tree

reweighted simulated data as seen in Figure 12. There was highly consistent performance across the 20 runs of the code when compared to other methods as shown by the graphs in Figures 7 and 8. This can be attributed to the low variance error given by the method, as each set of 'real' data should statistically be the same, as should each set of simulated data. The algorithm's performance at reweighting one of these distributions to look like the other should be consistent across all runs of the algorithm if that algorithm has low variance.

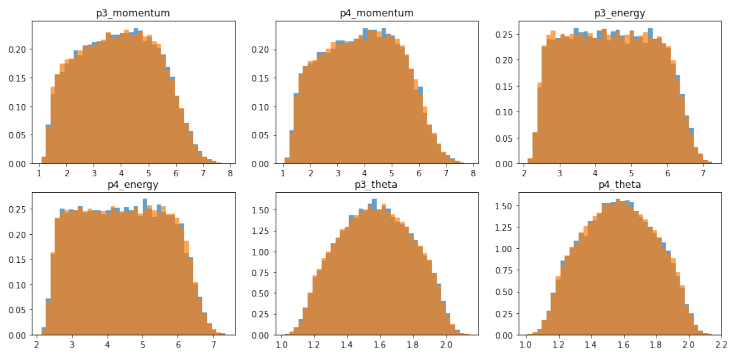

Figure 12. Simulated data reweighted using folding reweighter

Once the folding GB reweighter was found to be the best performing algorithm for a collision with a two body final state, the complexity of the data was increased by simulating a decay in one of the particles created in the collision. This three body final state then had 9 variables for the models to reweight and the folding GB reweighter could be tested to ensure it was scalable to more complex problems. The collision data with no weights applied is shown in Figure 13, and with weights calculated by the folding reweighter in Figure 14.

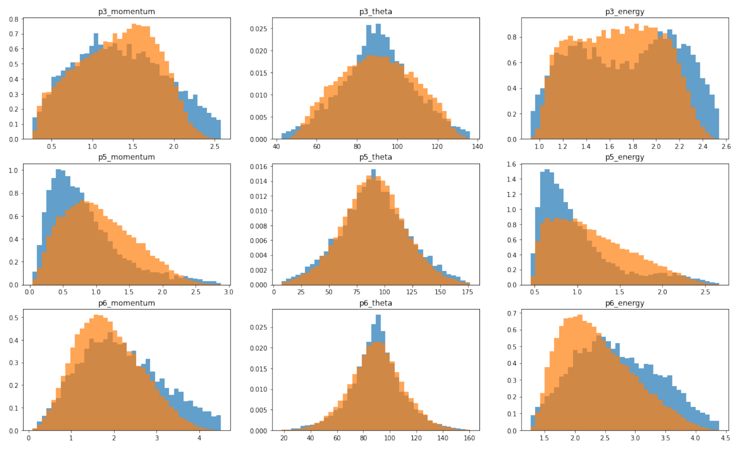

Figure 13. Unweighted 3 body final state features

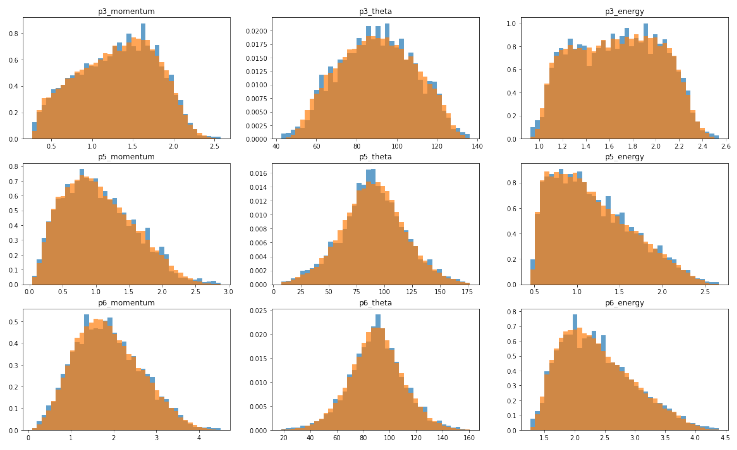

Figure 14. 3 body final state reweighted using folding reweighter

The performance of the folding reweighter was also tested against the standard GB reweighter and a neural network reweighter on this 3 body collision data, and it again gave the best results as seen in Table 2.

	KS Statistic	KL Divergence
No Weights	0.12592 ± 0.004791	0.085426 ± 0.003350
GB Weights	0.040462 ± 0.004053	0.061617 ± 0.01159
Folding GB Weights	0.018880 ± 0.001118	0.022419 ± 0.004803
Deep Adam NN	0.054736 ± 0.007150	0.064276 ± 0.006794

Table 2. KS and KL results from the reweighting of three body final state collisions, ± SD

8. Conclusion

It was found that a reweighter based on gradient boosted regression trees, used on data with a folding algorithm, could be applied to one data set and weight it to have a distribution significantly more similar some target data set. This method continued to be successful when the complexity of the data was increased so it is expected that it could be used when reweighting simulated collision data to appear more like some real data measured in some detector experiment, making the simulated data more useful for applications such as detector calibration or for use in training event identification algorithms. For further research, if some real collider data could be obtained then the algorithms could also be tested on this data. Additionally research could be conducted on the usefulness of other forms of neural networks than multi layer perceptrons in reweighting algorithms. There are more advanced network architectures that should be tested and may give better performance than the MLPs in this project.

References

1 Sjöstrand T, 2012, *Introduction to Monte Carlo Techniques in High Energy Physics* CERN Summer Student Lecture Part 1

2 Martschei D et al 2012 J. Phys.: Conf. Ser. **368** 012028

3 Burkov A, 2019, *The Hundred Page Machine Learning Book* (Quebec City: Self-Published)

4 Gupta P 2017 *Cross-Validation in Machine Learning* Towards Data Science

5 Castro R, *The Empirical Distribution Function and the Histogram* Eindhoven University of Technology

6 Kolmogorov test. Encyclopedia of Mathematics. URL: `http://www.encyclopediaofmath.org/index.php?title=Kolmogorov_test`

7 NIST/SEMATECH, 2013, e-Handbook of Statistical Methods, URL: `http://www.itl.nist.gov/div898/handbook/`

8 Kullback-Leibler information. Encyclopedia of Mathematics. URL: `http://www.encyclopediaofmath.org/index.php?title=Kullback-Leibler_information`

9 Theodoridis S, 2015, *Machine Learning A Bayesian and Optimization Perspective* (Amsterdam: Elsevier)

10 Agarap A 2018 *Deep Learning using Rectified Linear Units (ReLU)* Adamson University

11 Swingler K 2012 *Computing and the Brain* University of Stirling

12 Kingma D and Ba J 2014 *Adam: A Method for Stochastic Optimization* 3rd International Conference for Learning Representations

13 Nocedal J 1980 *Updating quasi-Newton matrices with limited storage* Math. Comp. **35** 773-82

14 Nielsen M 2015 *Neural Networks and Deep Learning* (San Francisco: Determination Press)

15 Wu X et al 2007 Knowl Inf Syst **14** 1–37

16 Schapire R E 2001 *The Boosting Approach to Machine Learning, An Overview* MSRI Workshop on Nonlinear Estimation and Classification

17 Schapire R E, 2014, *Boosting : Foundations and Algorithms* (Cambridge, Massachusetts : MIT Press)

18 Loh W 2011 WIREs Data Mining Knowl. Discov. **1** 14–23

19 Rogozhnikov A 2016 J. Phys.: Conf. Ser. **762** 012036

19 Hastie T Tibshirani R Friedman J 2009 The Elements of Statistical Learning (2nd ed.) (New York: Springer)

20 Friedman J 1999 Computational Statistics & Data Analysis **38** 367-78

21 Rokach L and Maimon O 2005 IEEE Transactions on Systems, Man, and Cybernetics **35** no. 4 476-87

Python packages

(i) The Scikit-hep package: `https://scikit-hep.org/` was used in the collision simulations

(ii) The basis of the adaboost and neural network reweighting algorithms was the scikit-learn python machine learning package found at: `https://scikit-learn.org/stable/index.html`

(iii) The hep-ml package: `https://arogozhnikov.github.io/hep_ml/` was used for the gradient boosting and folding algorithms

Appendix A. Sample of code used

A sample of the python code used in this project is provided at: `https://colab.research.google.com/drive/1zCEjV2nLgVcyefHuRakdGv2BP4HGwvEY`

Appendix B. Full sized regression tree

A full visualisation of a regression tree used to classify this data for reweighting is provided here: `https://drive.google.com/file/d/1cyjUa7Lh7j10j49JBRYmzgAU-xal-_P4/view?usp=sharing`

Acknowledgments
I would like to extend my thanks to my supervisor Prof. Ireland for his valuable support and advice during the completion of this project.