JOINT INSTITUTE FOR NUCLEAR RESEARCH

Veksler and Baldin Laboratory of High Energy Physics

REPORT ON THE SUMMER STUDENT PROGRAM

Development of Machine Learning Approach to Secondary Vertices Reconstruction at the BM@N Experiment

Supervisors:

Oleg Rogachevskiy

Pavel Batyuk

Student:

Alexander Rogozin

Moscow Institute of Physics and Technology

Participation period:

July 2 – August 16

Dubna, 2017

Contents

Abstract

BM@N is a high energy physics experiment with a fixed target. Existing secondary vertex reconstruction process can be divided into three parts: hit reconstruction, tracking and finding secondary vertices from known tracks. The last step is performed by exhaustive search through pairs of tracks. Each pair is propagated in order to find a secondary vertex or to establish that the pair is "uninteresting", which means it was not born by a particle decay. In this work, machine learning was implemented to weed out "uninteresing" pairs of tracks. Several datasets, algorithms and feature sets have been explored.

1 Introduction

When heavy ion hits the target, several new particles are produced. Starting points of their tracks are called primary vertices. During the flight through the detector, some particles may decay, and a point where decay took place is called a secondary vertex. In order to find these vertices, every pair of tracks in event is considered and tracks are propagated with the usage of Kalman Filter.

Dropping out pairs of tracks which do not start at a secondary vertex will decrease the amount of work for Kalman Filter. Moreover, some events consist of dozens or even hundreds of tracks, and that is why this preliminary analysis is beneficial.

Another implementation of machine learning is filtering noise on the invariant mass histogram. If this histogram included only track pairs which refer to Λ^0 decay, there would be a high narrow peak. However, some uninteresting pairs are included in the histogram and the peak may not be seen because of noise. That is why using machine learning algorithms to filter the noise can increase signal to background ratio.

2 About BM@N

BM@N is an experiment with a fixed target which aims at studying ion collisions at energies up to 4 GeV. The set-up includes 6 GEM-detectors, however it is still under construction and in future will have 12 stations, which are orthogonal to Z axis (Fig.2). In order to establish particle's charge to mass ratio, curvature of its trajectory is needed. That is why particles knocked out of target by ion collision fly into a magnetic field.

Note: magnetic field is not homogeneous. For this reason, tracks cannot be approximated simply by circles. A field map is used and tracks are propagated by Kalman filter.

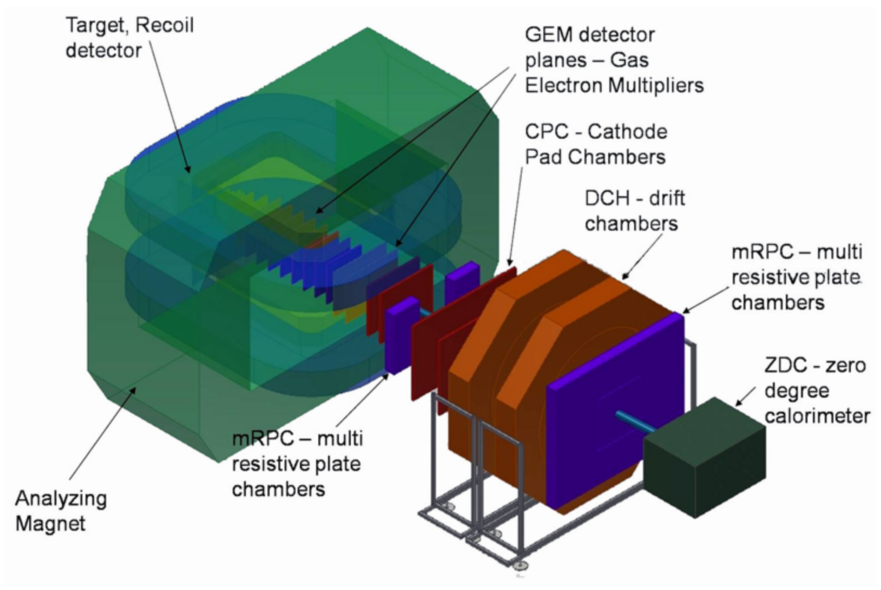

Target, Recoil detector
GEM detector planes – Gas Electron Multipliers
CPC - Cathode Pad Chambers
DCH - drift chambers
mRPC – multi resistive plate chambers
ZDC - zero degree calorimeter
mRPC – multi resistive plate chambers
Analyzing Magnet

Figure 1: BM@N structure

3 Machine Learning Basics

The most common problems in machine learning are classification, regression and clustering. Classification is refering an object to a particular class from a finite set, regression is reconstruction of a continuous mapping and clustering is finding classes of similar objects.

We will stick to classification.

3.1 Problem statement

Suppose we have a set $\{(x_i, y_i)\}_{i=1}^{n}$, where $x \in \mathbb{R}^n,\ y \in \{C_1, ..., C_n\}$ ($y \in \mathbb{R}$ for regression problem). x components are called *features* and y is called *target*. The aim is to reconstruct the mapping $y = f(x)$ in the best way (in a certain sense).

Note: not necessarily $f(x) \in \{C_1, ..., C_n\}$.

Note: if we are given corresponding values for each sample, it is called *supervised* learning. If not, *unsupervised* learning. Classification and regression refer to supervised learning and clustering is an unsupervised learning problem.

More strictly, a *loss function* $L(y, \hat{y})$ is introduced and a class of functions W is fixed. The solution for the classification problem is

$$\hat{f} = \underset{f \in W}{argmin} \sum_{i=1}^{n} L(y_i, f(x_i))$$

However, this is a problem of functional minimization, and it is opaque how to solve it numerically. That is when class W begins to matter. For example, if W is a class of linear functions, it brings us to the following problem:

$$\min_{w^1,\dots,w^n \in \mathbb{R}} \sum_{i=1}^{n} L(y_i, \sum_{i=1}^{n} w^j x_i^j)$$

Here $w^1, \dots, w^n$ are also called *weights*. This is minimization problem not for a functional, but for a function of n arguments $w^1, \dots, w^n$. It is possible to solve it numerically.

3.2 Quality Metrics

Model performance can be estimated with the usage of quality metrics. We will consider quality metrics for a binary classification problem, which means $y \in \{-1, 1\}$. Elements of class -1 are also called *condition negative*, elements of class 1 - *condition positive*, elements of class -1 which were classified by model correctly - *true negatives*, elements of class 1 classified as class -1 - *false negatives*, etc.

There are some common metrics:

$$Recall = \frac{\sum True\ Positive}{\sum Condition\ Positive}$$

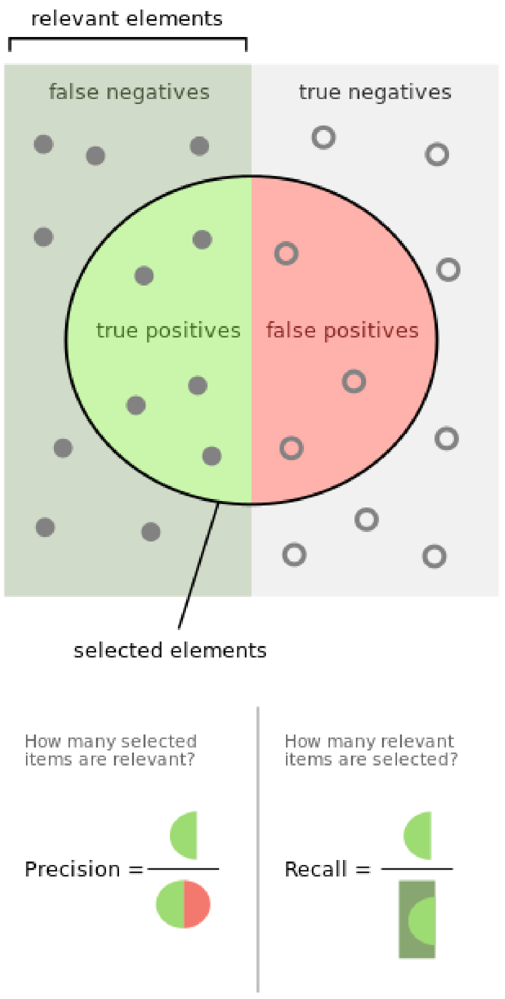


Figure 2: Precision and Recall

$$Precision = \frac{\sum True\ Positive}{\sum Predicted\ Positive}$$

$$Accuracy = \frac{\sum Correctly\ Predicted}{Total\ Set}$$

$$True\ Negative\ Rate = \frac{\sum True\ Negative}{\sum Condition\ Negative}$$

$$F1\ score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Quality metrics can describe model performance from different sides. For example, if we have a set where 40% samples are of class -1 and 60% are of class 1, then accuracy of 0.95 is a pretty good result. On the other hand, let class 1 consist only of 1% of total set. Then accuracy of 0.99 can be reached by simply classifying every sample as -1. In this case, high accuracy score does not mean classification quality is good.

3.3 Loss Functions

As stated above, we are looking for mapping $\hat{f}$ such as

$$\hat{f} = \underset{f \in W}{argmin} \sum_{i=1}^{n} L(y_i, f(x_i))$$

The process of finding an optimal $\hat{f}$ is called *model training* or *learning*. It is performed by some optimization process.

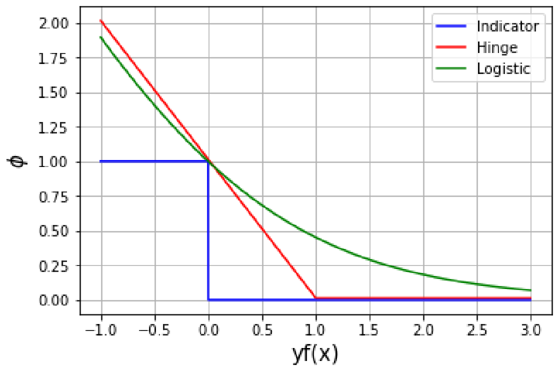


Figure 3: Loss Functions

We introduce a function $L(f(x), y) = \phi(yf(x))$. A natural choice for ϕ is a $0 - 1$ indicator function: $\phi(yf(x)) = 1$ iff y and $f(x)$ are of one sign. However, this function is not smooth and gradient-based optimization methods cannot be implemented.

In order to make the problem solvable for gradient-based optimization methods, smooth approximations of indicator function are implemented. Typical loss functions are:

Hinge loss: $\phi(t) = max\{0, 1 - t\},\ L(f(x), y) = max\{0, 1 - yf(x)\}$

Logistic loss: $\phi(t) = \frac{1}{\ln 2}\ln(1 + e^{-t}),\ L(f(x), y) = \frac{1}{\ln 2}\ln(1 + e^{-yf(x)})$

Note: loss function and quality metric are different. Quality metric is used to evaluate model's performance, but cannot be used for learning since it is not smooth (if we consider most common metrics).

3.4 Overfitting

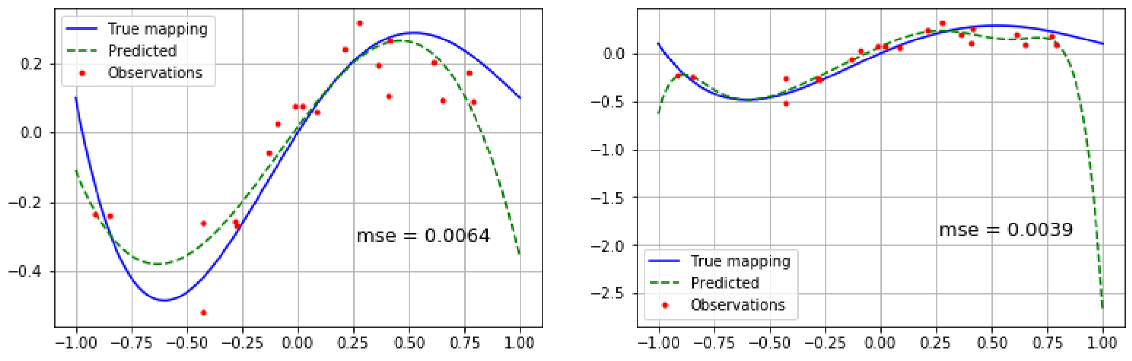
Consider our problem once again:

$$\min_{f \in W} \sum_{i=1}^{n} L(y_i, f(x_i))$$

The wider class W, the lower loss we can obtain. Wider class W means the model is more complex. However, complicated models may not generalize well. It can be shown by a following regression example.

Say we have a set of noised observations of scalar function values: $\{(x_i, y_i)\}_{i=1}^{n}$. We will seek the optimal reconstruction mapping in the class of polynoms:

$W_n = \{f(x) = a_0 + a_1 x ... + a_n x^n\}$ with a quadratic loss $L(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$. It is a least square method.



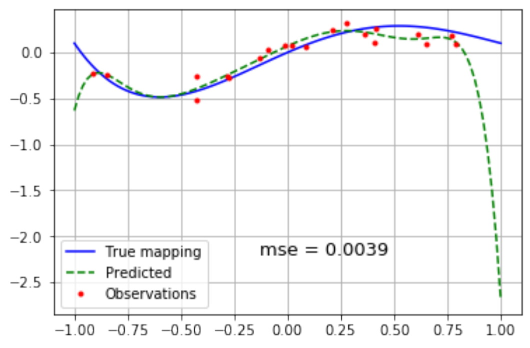
a) degree = 2, simple model b) degree = 8, complex model

Figure 4: Model complexity

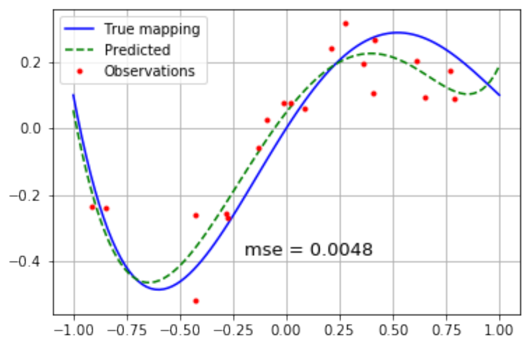
Class W_n becomes wider and model more complicated with the growth of n. More complex model does not catch the tendency and does not generalize well, *although it has lower mean squared error on the training set.* This effect is called *overfitting*. Informally, it means that model becomes too sensitive to local patterns in data without catching the global trend. More formally, training set can be thought of as a realization of a random value and model is a function which takes this set as an argument. Overfitting takes place because complicated models have high dispersion.

There are several ways to reduce overfitting. To decrease model's dispersion, a training can be performed on a bigger set. Moreover, a special technique called *regularization* can be implemented. We will consider the last one in detail.

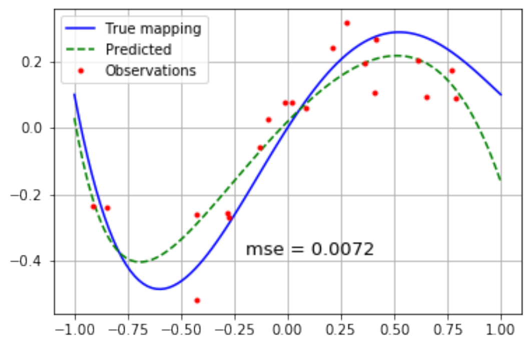
In the least square method we build a linear model over a set of features



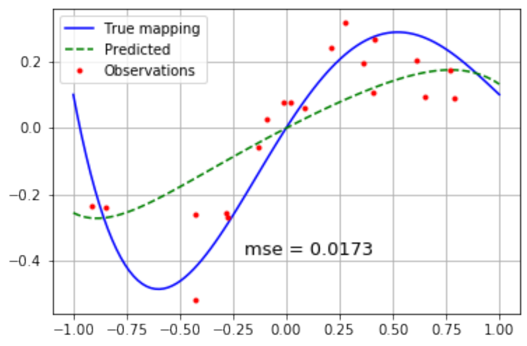
a) $\alpha = 0$



b) $\alpha = 0.01$



c) $\alpha = 0.1$



d) $\alpha = 1.0$

Figure 5: Regularization effect

$\{1, x, x^2, ..., x^n\}$ with a square loss function $L(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, where $\hat{y}_i = \sum_{j=0}^{n} w^j x_i^j$. The loss function can be rewritten as

$$L(y, w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \sum_{j=0}^{n} w^j x_i^j)^2.$$

Taking a look at Fig.4b, one can notice that model has a high weight at x^8. The idea of regularization is to forbid the model to have big weights. This is achieved by modification of the loss function. A *penalty* is added for each weight. Common modifications are:

L1 regularization, Lasso: $L(y, w) + \alpha\sum_{i=1}^{n}|w^j|$

L2 regularization, Ridge: $\hat{L}(y, w) = L(y, w) + \alpha\sum_{i=1}^{n}(w^j)^2$

Hyperparameter α is found empirically in most cases.

Regularization impact on model behavior is shown in Fig.5. Note that too strong regularization leads to underfitting.

3.5 Model Evaluation

If model performance is evaluated on the training set, the quality may be overestimated. That is why evaluation is performed on a different, *test* set. However, if model parameters are tuned to gain a bigger score on test set, it makes the model overfit to a particular dataset again. There are several approaches to adequate model evaluation.

1) *Validation set*. Data is divided into a train, validation and test sets. Learning is performed on the train set, model parameters are tuned to gain better performance on the validation set and final estimation is done on the test set.

2) *Cross validation*. Data is divided into n equal parts (*splits*, *folds*). Model is trained on splits $1, 2, ..., n-1$ and quality is estimated on the split n. After that, model is trained on splits $1, 2, ..., n-2, n$ and quality is estimated on split $n-1$, etc. Finally, obtained n scores are averaged. Moreover, deviation of model score can be found, which allows to estimate model's dispersion.

4 Problem Statement

A binary classification problem is considered. A pair of tracks is assigned to class 1 if it was born by a decay ($\Lambda^0 \to \pi^- + p$ in our case) and to class -1 otherwise. The aim of the algorithm is to learn to recognize whether the pair of tracks belongs to class -1 or to class 1.

4.1 Datasets

In order to obtain train and test datasets, we need to know for sure, which pair of tracks relates to a decay and which does not. That is why model training and quality evaluation was performed not on real data, but on Monte-Carlo simulations. A simulation consists of yielding particles out from a target, propagation of their trajectories and modeling the behaviour of the detector. Several types of particle generators have been explored. BOX generator was used to generate a fixed set of particles per event. Results of these simulations were then used to try different machine learning algorithms. Simulations with QGSM generator, however, are closer to reality and they were used for model evaluation.

4.2 Features

Every sample consists of parameters of the two tracks. Two different approaches to parameterization have been considered.

1) *Coordinate parametrization.*

Tracks are given by hits coordinates and momenta projections on each GEM-detector. The feature space has dimension

$2tracks \cdot 6hits \cdot (3coordinates + 3momenta\ projections) = 72.$

However, tracks may consist of 4, 5 or 6 hits. That is why some samples have missing features, which is the biggest problem of this approach to track parametrization.

This parametrization was used in order to weed out uninteresting pair of tracks, which reduces amount of work for Kalman filter.

2) *Cut parametrization.*

In this approach, pair of tracks is propagated by Kalman filter to the plane of primary vertex. The set of features was the following(Fig.6):
1) Z-coordinate V_0 of closest approach obtained through propagation in XZ and YZ planes;
2) Distance between tracks at the point of closest approach;
3) X-coordinates of tracks in primary vertex plane;
4) Momenta and curvature of each track at the point of closest approach.

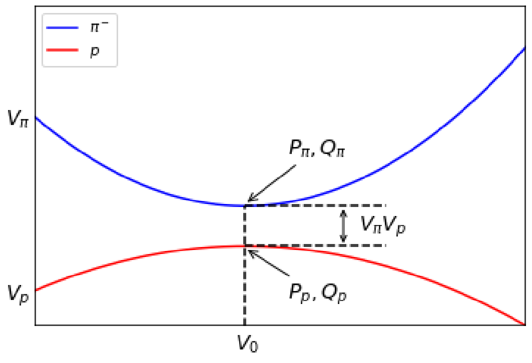


Figure 6: Cut parametrization

In this feature set no missing values can occure, which is not true for coordinate parametrization.

Cut parametrization was used to filter noise on invariant mass histogram and increase *signal to background* ratio.

4.3 Quality Metrics

Our aim is to weed out as many track pairs of class -1 as possible. However, Λ^0 decay is a rare phenomenon, and we do not want to lose track pairs which were born by such a decay. Speaking more formally, our model should have a good *true negative rate* and a high *recall* as well. This brings us to a *true negative rate - recall trade-off*. This trade-off will arise for several times below in this report.

5 K Neighbours Classifier

5.1 Method Overview

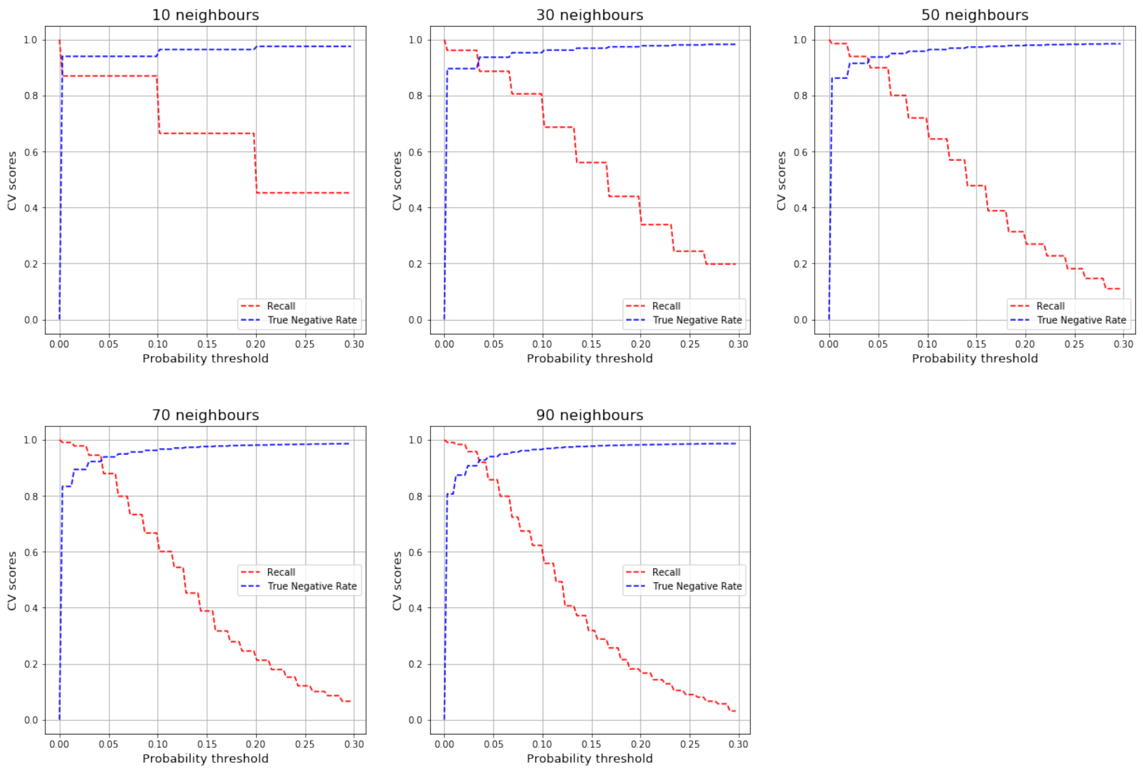
K nearest neighbours classifier is a simple classification algorithm. Given a training set $\{(x_i, y_i)_{i=1}^n\}$, natural k and a sample x_*, it finds k nearest (in a sense of specified metric) to x_* training samples. If m of them belong to class 1, the algorithm returns that probability of belonging x_* to class 1 is $\frac{m}{k}$. After that it is possible to establish a minimum probability to classify x_* as 1. For example, if threshold is 0.2 and algorithm returns a probability of 0.25, then a sample is assigned to class 1.

The reason for using this method is the following. Track parameters are feature vectors. If feature vectors are close to each other in a sense of Euclidean metric, it means that tracks are somehow similar. Consequently, two pairs are likely to refer to the same class -1 or 1 if their feature vectors do not differ much.

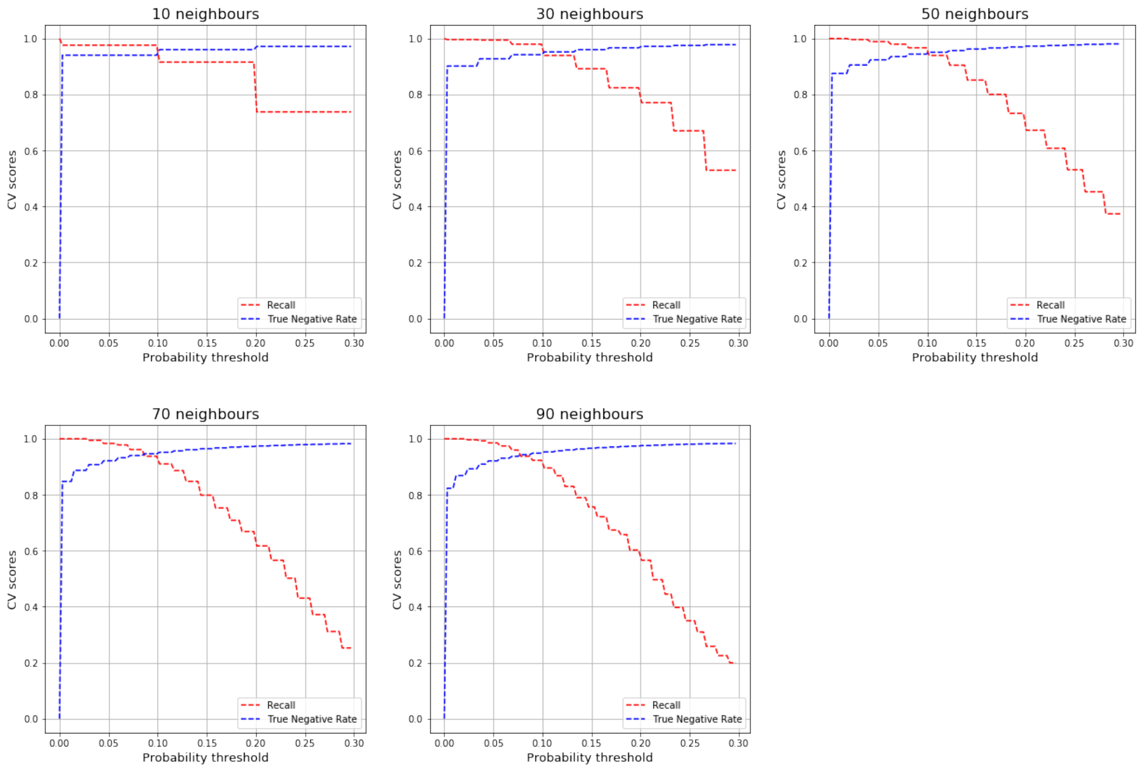
Note: KNN classifier counts distances between feature vectors and its work depends on the distances. Such algorithms are called *metric* methods. Non-metric methods are described further in this report.

5.2 Experiments

This method was implemented on datasets in coordinate parametrization (see section 4.2) simulated with BOX generator. At every event 6 particles



a) Raw data



b) Scaled data

Figure 7: KNN classifier results

were generated: $\pi^-, \pi^+, p, e^-, e^+, \Lambda^0$ and then 6-hit tracks were selected. The algorithm has 2 parameters: number of neighbours k and probability threshold. Model evaluation was performed on a 4-fold cross validation.

KNN is a metric method, which means features which have bigger values will have a higher impact on its decision. In order to eliminate this effect, feature scaling is used. Every feature in the training set undergoes a linear transform so that its mean and standard deviation over the training set become 0 and 1 respectively. After that, all features are "in more equal conditions".

When recall goes down, true negative rate increases. It is an illustration of true negative rate - recall trade-off mentioned above.

Experiment shows that feature scaling improves model performance significantly (Fig.7). However, KNN requires a fixed number of features, which means it can only work with tracks of fixed length, for example, 6.

It is elusive what to call the best result in this case. For example, if we establish a threshold of 0.98 for recall then the best result is true negative rate of 0.92 with the usage of 70 neighbours and if the recall threshold is reduced to 0.95, the best true negative rate is 0.97 at 90 neighbours. A possible way to evaluate quality is *area under ROC-curve* [4]. In this case, the best score is performed on 50 neighbours: 0.987 *roc auc*.

6 Gradient Boosted Trees

6.1 Method Overview

The core idea of gradient boosting is *ensembling* of *weak learners* or *base learners*. This means many simple models $\{m_1, ..., m_k\}$ are built and output of the ensemble on sample x_* is a weighted output of models: $Y(x) = \sum\limits_{i=1}^{k} w^i m_i(x_*)$.

Say we have 100 estimators with 60% accuracy each. It means every weak learner has a chance of 60% to be right about classifying a sample. Suppose that estimators make their predictions independently (which is not true in real cases, however). If we take weights $w^1 = ... = w^k = \frac{1}{k}$, then an average model output on a positive sample will be $0.2 > 0$ (-0.2 on negative sample), but the ensemble will have lower dispersion and consequently higher accuracy.

This method has been chosen because decision trees have an implementation which is able to work with missing values (see [7]).

6.1.1 Decision Trees

Decision tree has conditions at its internal nodes and decisions (-1 or 1) at the leaf nodes. Process of building such a tree is the following. On each step the data is split along some of the x axis. In other words, a condition $x^i < a$ is placed into an internal node.

This split has to be optimal in some sense. More specifically, subsets on the left and on the right branches should be relatively pure, which means they should consist of approximately one class. There are several ways to evaluate subset purity.

1) *Gini impurity*: $I_G = \sum\limits_{i=1}^{J} p_i(1 - p_i)$ where J is the number of classes, p_i is the proportion of class i. $I_G = 0$ if the set consists of one class.

2) *Information gain*: $I_E = -\sum\limits_{i=1}^{J} p_i \log_2 p_i$.

<u>Note</u>: decision tree is a non-metric method.

There are also several stopping criteria for the tree building process: maximum depth reached, impurity does not decrease below ε in the split, minimum number of samples at the node reached, etc.

6.1.2 Gradient boosting

Suppose we have a model F_m. Gradient boosting step improves it to a better model $F_{m+1} = F_m + \gamma_{m+1} h_m(x)$. Here the weak learner $h_m(x)$ is fitted to the residual $y - F_m(x)$ and $\gamma_{m+1} = \underset{\gamma}{argmin} \sum\limits_{i=1}^{n} L(y_i, F_m(x_i) + \gamma h_m(x_i))$.

If class H of weak learners is a class of differentiable functions, we can update the model according to the following rules:

$F_{m+1}(x) = F_m(x) - \gamma_{m+1} \sum\limits_{i=1}^{n} \nabla_{F_m} L(y_i, F_m(x_i))$

$$\gamma_{m+1} = \underset{\gamma}{argmin} \sum_{i=1}^{n} L\Big(y_i, F_m(x_i) - \gamma \frac{\partial L(y_i, F_m(x_i))}{\partial F_m(x_i)}\Big)$$

That is why it is called *gradient* boosting.

6.2 Experiments

All experiments have been carried out with the use of *xgboost* package [7]. Model evaluation was performed on a 4-fold cross validation.

6.2.1 Dataset

The method has been implemented on datasets in coordinate parametrization simulated with QGSM generator. Track curvatures per pair were of different sign (corresponding to particles of different charge). The dataset was constructed this way, because we are looking for Λ^0 decay products, which are π^- and p.

The dataset consisted of 280000 samples. However, only 1326 track pairs corresponded to Λ^0 decays, which is less than 0.5%. It is a problem with *unbalanced classes*. In order to deal with it, class 1 (which corresponds to Λ^0 decay) was assigned some *weight*. It makes the loss function increase higher when an element of class 1 is misclassified.

Moreover, absolute value of momenta for each track in a pair was added to the dataset.

6.2.2 Parameter tuning

Gradient boosted tree algorithm has many parameters, the tuning of which can increase model performance. Tuning of these parameters can be considered as another optimization problem. Two approaches have been explored: finding optimal value for each parameter while others are fixed, which is analogous to coordinate descent, and an exhaustive search through a specified parameter grid - so called *grid search*. The second approach is computationally harder but may lead to higher score.

The following algorithm parameters were tuned:

1) *weight* - weight assigned to class 1;

2) *max_depth* - maximum tree depth;

3) *n_estimators* - number of boosted trees;

4) *gamma* - minimum loss reduction required to make a further partition on a leaf node of the tree. The higher *gamma*, the stronger the regularization;

5) *min_child_weight* - minimum weight needed in a child node of a tree;

6) *colsample_bytree* - percentage of features used for building each tree;

7) *subsample* - percentage of samples used for building each tree;

Other parameters including *learning rate* remained initialized by default.

As mentioned above, main metrics for evaluation are recall and true negative rate. We establish a threshold for recall 0.95 and try to obtain as high true negative rate as possible with this constraint.

In coordinate descent, features were considered in order mentioned above. This is an order of decreasing importance.

Note: the above statement is a heuristic.

Final results obtained by coordinate descent:

$Recall : 0.956$
$TNR : 0.776$
$weight : 1000$
$max\_depth : 4$
$n\_estimators : 71$

$gamma : 0$
$min\_child\_weight : 1$
$colsample\_bytree : 1.0$
$subsample : 0.8$

Note that tuning of *gamma*, *min_child_weight* and *colsample_bytree* did not have any impact.

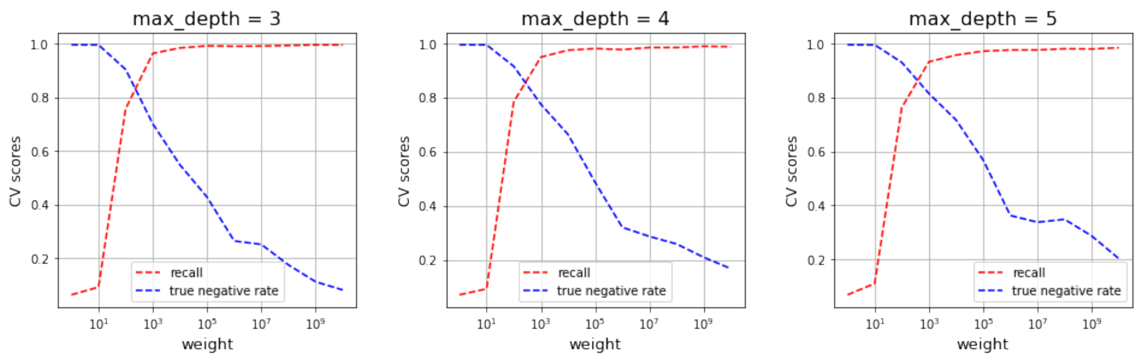
In grid search, the following grid of parameters was explored:

$weight : [500, 1000, 1500, 2000, 2500]$
$max\_depth : [2, 3, 4, 5]$
$n\_estimators : [20, 40, 60, 80, 100, 120, 140]$
$subsample : [0.6, 0.75, 0.9, 1.0]$
$colsample\_bytree : [0.6, 0.8, 1.0]$

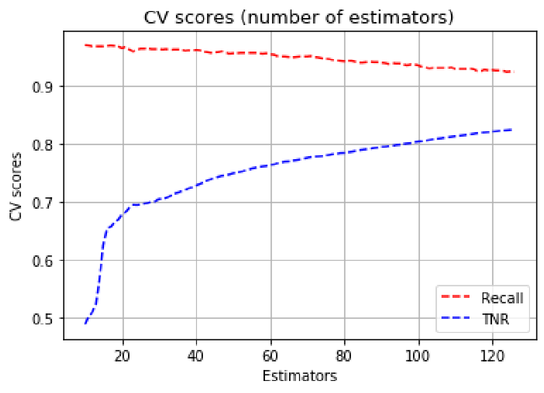
Final results obtained with grid search:

$Recall : 0.951$

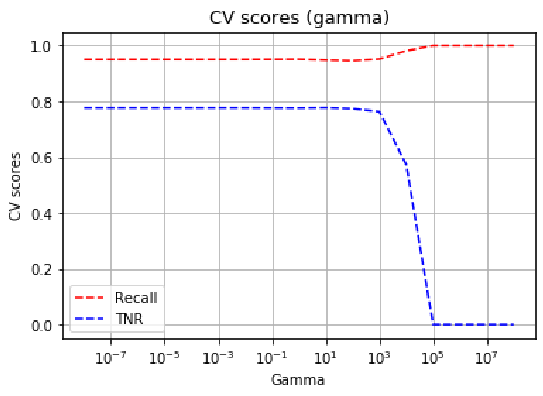
$gamma : 0$



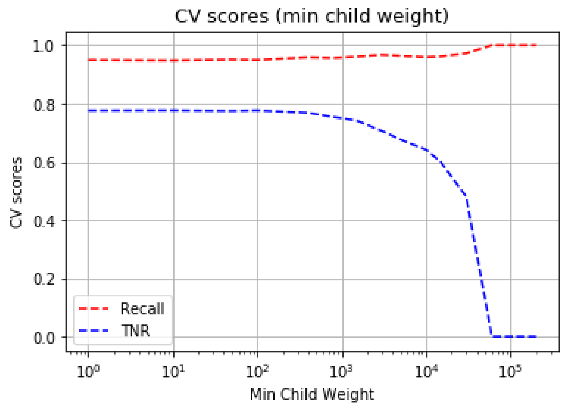
a) Tuning *weight* and *max_depth*



b) Tuning *n_estimators*



c) Tuning *gamma*



d) Tuning *min_child_weight*

Figure 8: XGBoost parameter tuning. Coordinate descent.

$TNR : 0.799$	$min\_child\_weight : 1$
$weight : 1000$	$colsample\_bytree : 0.6$
$max\_depth : 4$	$subsample : 0.9$
$n\_estimators : 100$	

One can see that TNR obtained with grid search is slightly better that TNR obtained with coordinate descent, which means coordinate descent heuristic worked fairly well.

7 Neural Network

In order to build invariant mass plot, pairs of tracks of opposite curvature are considered. For each pair cuts and invariant mass are calculated (see section 4.2). After that, some primary filtering is performed, i.e. tracks which have too high $v_\pi v_p$ are weeded out. The aim of neural network is to perform further filtering of noise pairs.

The calculation of cuts can be performed either on simulated or reconstructed tracks. Both approaches have been explored.

7.1 Method overview

Neural network is a biologically inspired algorithm architecture. A basic unit of this structure is a neuron.

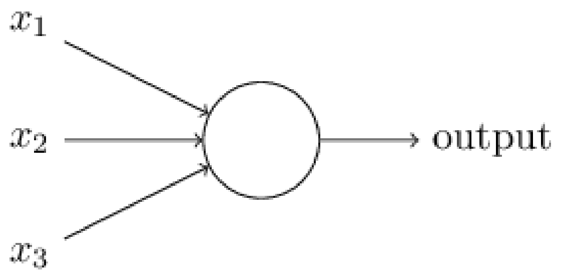


Figure 9: Neuron

Each neuron takes input data, transforms it in a particular way and passes the result to the next neurons. More specifically, for a given set of inputs $x_1, ..., x_m$, a neuron generates an output $\sigma(\sum_{i=1}^{m} w_i x_i + b)$ where w_i are *weights*, b is *bias* and σ is *activation function*.

A neural network consists of several neurons. We will stick to a *multilayer* of *sequential* network, which means neurons in it are organized into a number of layers, and output of each layer is an input for the following one.

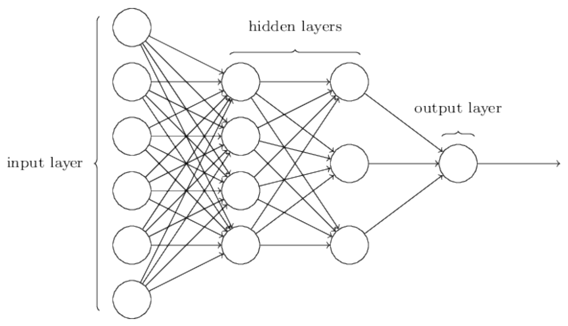


Figure 10: Sequential neural network

There are several common activation functions:

$$Linear: \ \sigma(x) = x$$
$$Sigmoid: \ \sigma(x) = \frac{1}{1+e^{-x}}$$
$$ReLU: \ \sigma(x) = \begin{cases} 0, \ x < 0 \\ x, \ x \geq 0 \end{cases}$$

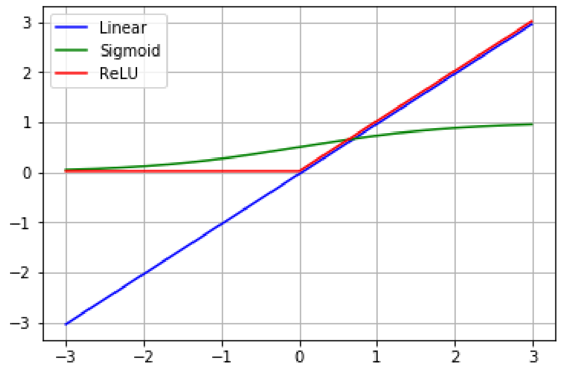


Figure 11: Activation functions

7.2 Experiments

7.2.1 Simulated data

Generation a dataset consisted of 3 steps:
1) QGSM simulations;
2) Calculating cuts for track pairs;
3) Weeding out pairs with inappropriate cuts.

The dataset consisted of 9200 noise pairs and 8500 pairs which referred to Λ^0 decay. This data was obtained from $720,000$ events simulated with QGSM generator. Data was divided into train and test sets consisting of 80% and 20% of the whole data respectively, and validation set constituted 20% of the train one.

Model performed training accuracy of 0.977, validation accuracy of 0.970 and test accuracy of 0.973, which means it did not overfit. However, noise filtering is not notable on the plot of the whole data, since Λ^0 constitute about 50% of the set, which is fairly much (Fig.12 a, b). In order to simulate real amount of Λ^0 in the set and make the model performance more visible, number of Λ^0 was artificially reduced by 20 times. After that, the impact of neural network filtering became more evident (Fig.12 c, d).

7.2.2 Reconstructed data

It is important to make a difference between cuts obtained from *simulated* data and cuts obtained from *reconstructed* data.

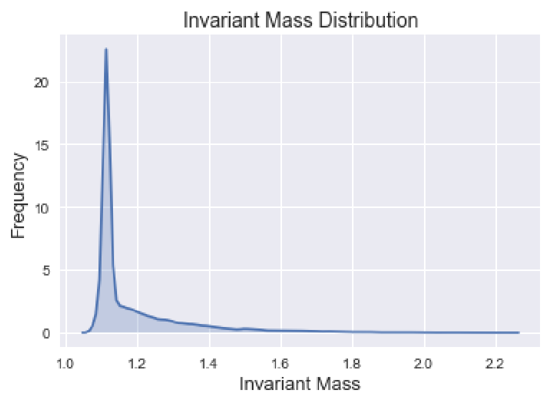
Generation a dataset consisted of 4 steps:
1) QGSM simulation;
2) Reconstruction;
3) Calculating cuts for track pairs;
4) Weeding out pairs with inappropriate cuts.

The dataset consisted of 8700 noise samples and only 80 samples corresponding to Λ^0 decays. This brings us to a problem with unbalanced classes again.

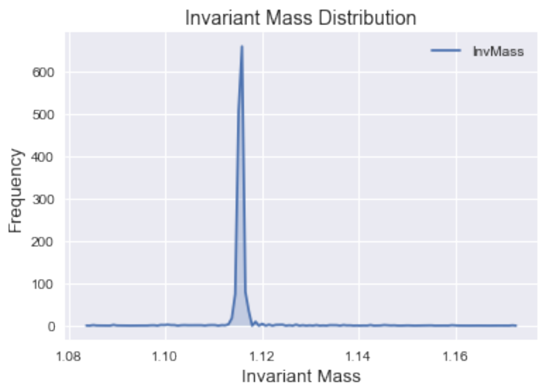
An attempt to filter this data using a model trained on simulated dataset did not give good results - recall of 0.125 (Fig.12 e, f). This happened because the training set consisted of simulated data which is different from reconstructed. More formally, the neural net learnt a mapping in one area of feature space and was tested on another one. It led to a necessity of training a model on reconstructed data.

However, 80 samples of the interesting class is extremely few: a bigger dataset was required.

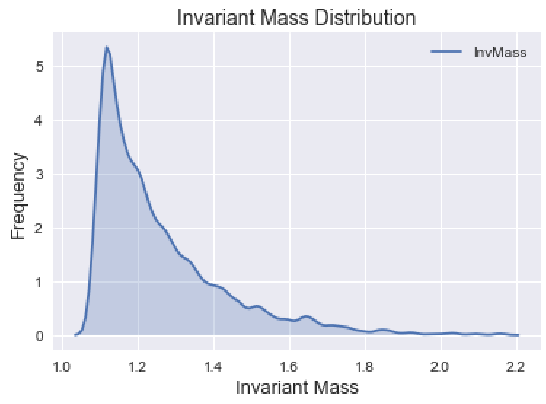
One way is to run simulations with QGSM, reconstructions and cut calculations once again. Taking into account that these 80 Λ^0 decays have been obtained from $720,000$ events and we want about 1000 decays, another $10,000,000$ QGSM simulations are needed. This is very computationally



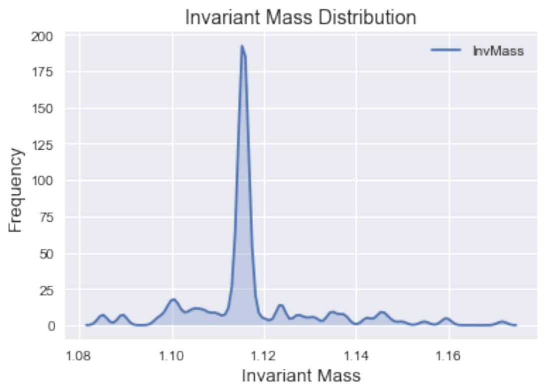
a)



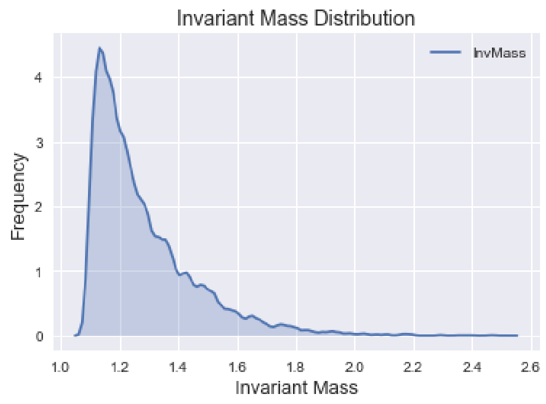
b)



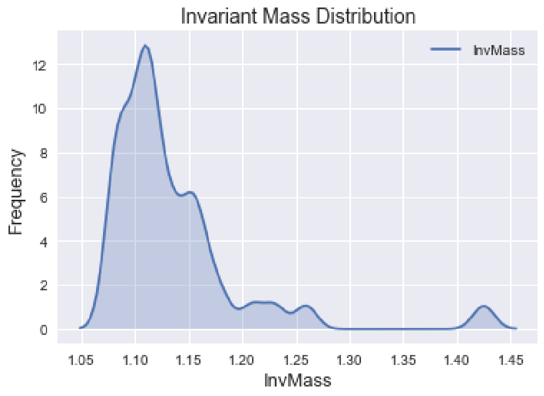
c)



d)



e)



f)

Figure 12: Invariant mass plots. a) Simulated data before filtering; b) Simulated data after filtering; c) Simulated data with reduced amount of Λ^0 before filtering; d) Simulated data with reduced amount of Λ^0 after filtering; e) Reconstructed data before filtering; f) Reconstructed data after filtering.

expensive as well as memory consuming.

Another way is to artificially add information about Λ^0 decays into the training set basing on some statistics we already have. This approach is called *oversampling*. More specifically, we could generate only Λ^0-s with BOX generator. However, data about Λ^0-s has some specific distribution over the dataset. It can be thought of as a random vector. If we generate random Λ^0-s with BOX generator, we may get inadequate oversampling (Fig.13). That is why we have to find a way to add samples which correspond to a true distribution.

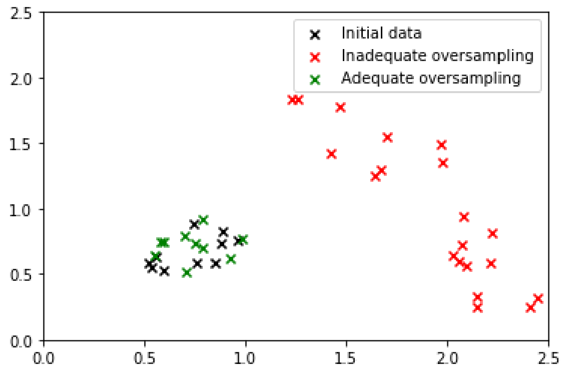


Figure 13: Oversampling

It would be ideal to generate Λ^0-s which correspond to a true physical distribution and which will be finally found in *reconstructed* data. However, we have only 80 samples from this distribution, which is too few. Instead, we can generate Λ^0-s which will be found in *simulated* data, because in this case we have more statistics - 8500 samples.

This approach is computationally cheaper than running the whole cycle of dataset generation beginning with QGSM simulations. Indeed, we obtained 8500 Λ^0-s from $720,000$ simulated events and 80 Λ^0-s from $720,000$ reconstructed events. Say we want 1000 Λ^0-s. If we use QGSM simulations, we will need about $720,000 \cdot \dfrac{1000}{80} \approx 10^7$ runs. But if we yield appropriate Λ^0-s with BOX generator, only about $8500 \cdot \dfrac{1000}{80} \approx 10^5$ runs are required.

The problem is stated as following. We have a set of particles and our aim is to enlarge this set by generating new particles *with the same momenta and angles distribution at the primary vertex*. Strictly speaking, we need to take the primary vertex coordinates distribution into account, as well, but more than 99% of samples in the set (a set of 8500 Λ^0-s found in reconstructed data is meant) have primary vertex at $(0, 0, 0)$.

We will stick to the following plan:
1) Estimate probability density of the existing set;
2) Generate a new larger set with this probability density;
3) Pass this set to the BOX generator and run simulations;
4) Run reconstruction;
5) Calculate cuts for tracks pairs;
6) Weed out track pairs with inappropriate cuts.

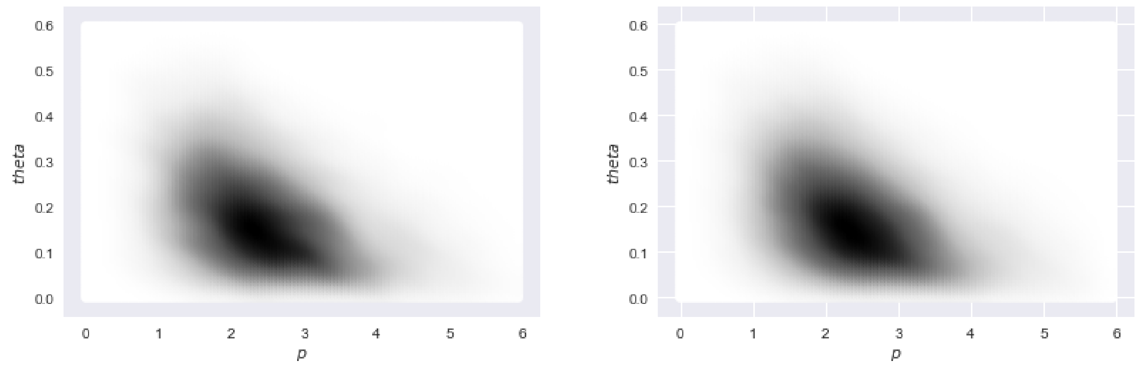
Λ^0-s have 3 parameters - momenta p and angles φ, θ. From the physical considerations, ϕ has a uniform distribution over $[0, 2\pi]$ and is mutually independent from p and θ. The aim is to boost a set consisting of p and θ and then add a set of uniformly distributed φ to it.

Density evaluation was performed by *kernel density estimation* ([2], [3], [8]). Obtained density is a function which is not set analytically, but can be computed at any point.

After that, a new set was computed in the following way.
1) Divide $p - \theta$ plane into equal rectangles;

2) Compute density at middle point of each rectangle;
3) Add points to each rectangle. A number of points is proportional to the density and the coordinates are distributed uniformly over the rectangle.

Density analysis has shown that this heuristic works fairly well (Fig14).

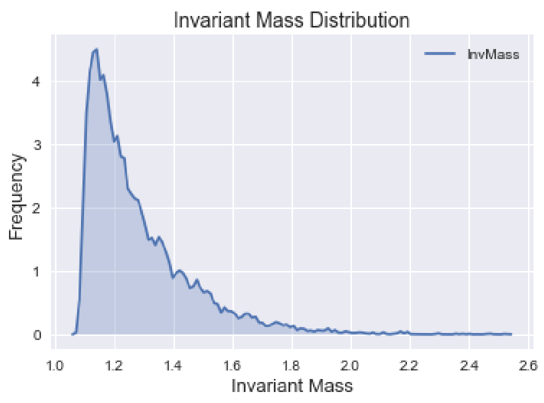


a) Initial dataset, 8500 samples b) Generated dataset, 10^5 samples

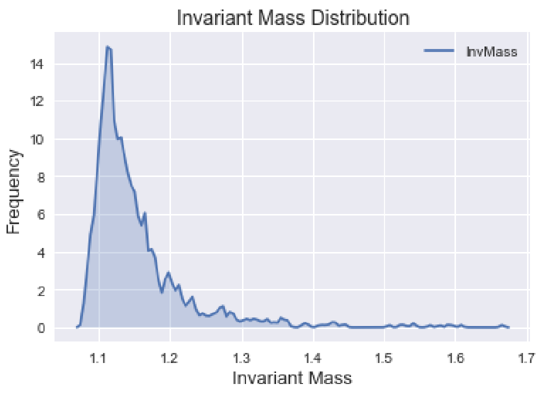
Figure 14: Density estimation

Now we can train a network on reconstructed data with oversampled amount of Λ^0-s. The model performs 0.94 recall and 0.85 accuracy. The results of filtering are shown on Fig.15.

Filtering performed on simulated data was more successful than one implemented on reconstructed data. This happens due to the fact that reconstructed data has a strong class imbalance.



a) Before filtering



b) After filtering

Figure 15: Noise filtering on invariant mass plots

8 Conclusion

8.1 Results

Several methods, approaches and tracks parametrizations have been explored. Obtained results are briefly listed below.

1) KNN, coordinate parametrization, only 6-hit tracks: roc auc 0.987;

2) Gradient boosted trees, coordinate parametrization: recall 0.951, true negative rate: 0.799;

3) Neural network, cut parametrization, simulated data: accuracy 0.973, recall 0.980, notable noise filtering (Fig.12 c, d);

4) Neural network, cut parametrization, reconstructed data: accuracy 0.850, recall 0.940, lower performance than in item 3 (Fig.15).

8.2 Possible continuation

Specificity of this problem is high class imbalance. In order to build a model, oversampling of smaller class is required. An attempt to perform this oversampling by generating Λ^0-s with BOX generator has been undertaken. There exist some strategies of oversampling the initial dataset synthetically [5]. These heuristics can be combined with the method described in this work.

The existing cuts filter 85% noise but also weed out about 30% of Λ^0-s. A possible solution is to imply machine learning to data not processed with cuts. However, this leads to a even higher class imbalance.

9 Acknowledgements

The author is grateful to Pavel Batyuk for his help with computer programming, including guide into BM@N software, working on remote server and debugging. I am obliged to Sergey Merts for ideas he proposed, as well. The author is also grateful to Oleg Rogachevskiy, without whom this summer program would not take place.

References

[1] Friedman, J. H. "Greedy Function Approximation: A Gradient Boosting Machine."

[2] Rosenblatt, M. (1956). "Remarks on Some Nonparametric Estimates of a Density Function". The Annals of Mathematical Statistics.

[3] Epanechnikov, V.A. (1969). "Non-parametric estimation of a multivariate probability density". Theory of Probability and its Applications.

[4] Fawcett, Tom (2006). "An introduction to ROC analysis, Pattern Recognition Letters"

[5] Chawla, Bowyer, Hall, Kegelmeyer (2002). "SMOTE: Synthetic Minority Over-sampling Technique"

[6] http://neuralnetworksanddeeplearning.com

[7] https://xgboost.readthedocs.io/en/latest/

[8] https://docs.scipy.org/doc/