Ultrafast sparse binning clustering enhanced by ML track scoring

Yuval Reina, Tel-Aviv Israel, Yuval.reina@gmail.com (Yuval.reina@gmail.com)

Trian Xylouris, Frankfurt am Main Germany, t.xylouris@gmail.com (t.xylouris@gmail.com)

August 2018

Competition Name: TrackML

Team Name: Yuval & Trian

Private Leaderboard Score: 0.80414

Private Leaderboard Place: 7

Summary

We use sparse binning to perform ultrafast clustering. Tracks are first chosen according to their length, and later are scored and merged using a Machine Learning (=ML) algorithm. At the final stage the tracks are expanded by adding the closest hits to the track. We describe our method in Chapter 1. In Chapter 2, we complete one full, yet quick, code run for training event 1000, which yields a 0.75 score.

The biggest advantage of using clustering by sparse binning is speed. This method can score 0.5 in just 40 sec using python on a single core, and it could be much faster if it was written in C++ and even faster by using parallel computing on a CPU or GPU. **The computational complexity of the clustering part in the algorithm is O(N)**. The feature calculation and binning can be done for every hit independently of the other hits and all hits are needed only for counting the number of hits in every bin (the Python implementation uses np.unique which is actually O(N logN)). We believe that with a careful implementation in C++ a score of 0.5 can be achieved in less than 10 seconds, allowing this algorithm to be an essential step in every fast algorithm (we will explain this statement later).

The second stage in the algorithm is ML merging. The fastest way to merge 2 solutions is by assigning to each hit the track with the highest number of hits. This method is used in the clustering main loop. However, the number of hits is not always a good indication for a good track. The ML algorithm uses various features which describe the track and is able to distinguish between good and bad tracks. Unlike other ML solutions presented in the TrackML Kaggle competition, our ML algorithm does not check the helix itself, as the clustering part already takes care of this.

The last part in the algorithm is quite straightforward: track expanding is done by selecting long tracks and adding to these tracks the closest loose hits (i.e. hits from short tracks).

The full solution is then:

- Run clustering a few times
- Use ML to merge

- Expand

In the solution we submitted, we used about 7 runs of clustering, each using 100000 loop iterations, merged and expanded twice (the 2nd time was just to add about 0.003 to the score by using a loophole in the definition of the competition metric (1) (https://www.kaggle.com/c/trackml-particle-identification/discussion/60638#354053))

Chapter 1: Methods and Results

Clustering using Sparse Binning

The basic idea of clustering it to relate to every hit a set of numbers (features) and then to group together hits whose features are close to each other. The first stage in clustering is finding features which describe a track well and which we can calculate separately for each hit. We have published a post on criteria for good features (2) (https://www.kaggle.com/c/trackml-particle-identification/discussion/61590).

Features

Due to the magnetic field in the experiment the particles form helixes, which travel along the z-axis. Each hit in the x-y-z space can be the member of any particular helix in a certain family of helixes. The idea of the algorithm is to go over each possible member of this family of helixes, and look at how many of all hits can belong to this particular member. The hits are then assigned to the helix (=track) with the most possible hits.

We start by assuming the particle is formed in a small cylinder around the origin i.e. the approximate starting point of the helix is $(0, 0, z0)$. According to the introduction papers, we usually have $|z0| < 5.5mm$.

This kind of helix can be defined by 3 numbers: its radius, its tangential angle at the origin in the xy plane (=the direction of the particle when it is created, in xy plane) and the slope of the helix (how fast the particle moves in the z direction compared to its velocity in xy).

Let

R = helix radius,

θ = tangential angle in the xy plane,

ϕ = slope,

(px, py, pz) = particle's initial momentum.

Note that

$$\theta = \arctan \frac{py}{px},$$
$$\phi = \arctan \frac{pz}{\sqrt{px^2 + py^2}}$$

For each hit (x, y, z) we can calculate its values for θ and ϕ, given R and $z0$. We define

$$kt = \frac{1}{2R}$$

We will iterate over kt. If $kt > 0$, then we deal with particles rotating clockwise and $kt < 0$ for counter-clockwise rotation. Using sum trigonometry, we get:

$$\theta\_ = \arctan \frac{y}{x}$$

$$rr = \sqrt{x^2 + y^2}$$

$$\Delta\theta = \arcsin\left(kt \cdot rr\right)$$

$$\theta = \theta\_ + \Delta\theta$$

(At this stage we ignore particles that rotate more then π radians)

$$\phi\_ = \arctan \frac{(z - z0) \cdot kt}{\Delta\theta}$$

θ and $\phi\_$ will be our features, but we need to tweak them a little to become useful.

First, we are only interested in the value of θ modulo 2π. Thus, instead of θ, we use the two features:

$$sint = \sin(\theta)$$
$$cost = \cos(\theta)$$

$\phi\_$ on the other hand is in the range $[-\pi/2, \pi/2]$, and does not have the above issue, but $\phi\_$ distribution is far from being uniform (for an extensive discussion about its distribution look here (3) (https://www.kaggle.com/c/trackml-particle-identification/discussion/63250#latest-371940)).

The solution we found suitable is to use the following instead of $\phi\_$:

$$\phi = \arctan \frac{(z - z0) \cdot kt}{3.3 \cdot \Delta\theta} \frac{2}{\pi}.$$

Sparse Binning Clustering

Fix a pair $(kt, z0)$. We calculate the three afore-mentioned features for each hit. Now, binning is the easiest way to cluster together different hits, based on whether its features are near each other.

To illustrate how binning works, suppose that for all hits, a certain feature of those hit lies always in the range $[-1, 1]$. We may then divide this interval into 20 bins of equal size 0.1:

$$[-1, -0.9), [-0.9, 0.8), \ldots, [0.9, 1].$$

We assign a number 0 , ..., 19 to every bin. If we have three features for each hit, then we assign to each hit a tuple with three values: One bin number for each of its three features:

```
 Hit #1 – (12,15,0)
Hit #2 – (11,5,2)
...
```

Hits which have the exact same three values are clustered together to form one track candidate.

The number of bins for each feature, as well as the size of each bin, may vary. In our Sparse Binning, we use a very simple binning algorithm. We start by using uniformly spread bins which can be easily calculated:

For feature F_j in the range $[-1, 1]$, we choose to have $2 * k_j$ bins, and define $B_j = int(F_j * k_j)$.

To combine several features together, and get the cluster id (=track_id), we just multiply each B_j with a different, large enough, number, and add the resulting values. For instance, if we have two features F_1, F_2 and we want to have $2 * k_1$, $2 * k_2$ bins for each feature, we can define:

$$(a) TrackId = int(F_1 * k_1) + (2 * k_1 + 1) * int(F_2 * k_2)$$

And that's it. Clustering is finished!

This calculation takes about 4mSec for 120000 hits on the Kaggle kernel platform compared to about 950mSec for dbscan, which is the most popular clustering algorithm used by the competitors in the TrackML challenge.

This type of clustering moves the bottleneck of the full solution form clustering to merging (will be discussed in the next section).

One weakness of the binning algorithm is its sensitivity, caused - among other things - by the hard borders between bins. As an example, let's take two hits:

```
 Hit1 – with features (-1e-7,0.1,0.3)
Hit2 – with features (1e-7,0.1,0.3)
```

If we use the calculation above, Hit1 and Hit2 will never be clustered together, no matter how many bins we have. This issue is solved by adding a random number r_j in the range $[0, 1)$ to every feature in the binning calculation, turning (a) to:

$$(b) TrackId = int(F_1 * k_1 + r_1) + (2 * k_1 + 1) * int(F_2 * k_2 + r_2).$$

Here, r_1, r_2 are random numbers with uniform distribution in the range $[0, 1)$. We change them for every pair $(kt, z0)$.

When compared to dbscan, sparse clustering has one big disadvantage, it's sensitivity. It can miss some of the hits because they fall in another bin. This sensitivity is also its strength, as it doesn't add many wrong hits to a track (outliers).

Simple Merging

Every time we perform a clustering, using a pair $(kt, z0)$, every hit gets a new track_id, thus generating one new list with track ids. We need to merge two lists at a time. The most efficient way we found is to measure the length of each track in the two lists, and let every hit choose the longest track it can.

If Hit #1 has track_id=1 in list #100 and track_id=10000002 in list #101 (the track_ids in both lists should be completely different) and in list #100 there are 10 hits with track_id=1 while in list #101 there are 7 hits with track_id=10000002, then hit#1 will be assigned to track_id=1.

To do this, we need to measure the track's length. The fastest way we know to do this in Python is by using numpy.unique. This operation takes 11mSec for 120000 hits on the Kaggle kernel platform. The numpy.unique computational complexity is O(NlogN), because it sorts the values. However, if we have enough memory, this task can easily be done in O(N). Also, we can reduce the amount of memory needed by hashing.

In our main loop we measure the length of a track twice. The first time in order to create a track candidate and the second time to update the final track length for each track (and their related hits), as the latter is used to merge the next list of track ids onto the current one.

The double track length measurement makes this the bottleneck of the algorithm.

Clustering – the full algorithm

The algorithm iterates over randomly selected pairs $(kt, z0)$. For every pair, the hits' features are calculated, and clustering is performed by the sparse binning algorithm. The new clustering result is then merged with previous ones with the merging algorithm described above.

Every 500 loops an extra step is taken to ensure that the following two conditions are met:

- at most 1 hit from the same detector (equal Volume_id, Layer_id, and Module_id) can belong to one track
- at most 2 hits from the same layer (equal Volume_id and Layer_id) can belong to one track We remove hits, until those condition are met. The hits, which are removed, are selected based on how far their features are from the means of the other hits' features.

In this last step, the algorithm also carves out "good" tracks which are long enough (hits won't be taken away or added to this track anymore).

While running the algorithm, we change the number of bins per feature and the minimal length for tracks to be carved out. These settings are user-defined parameters.

Clustering results:

The more pairs $(kt, z0)$ we iterate over, the better the final score achieved by clustering (below scores measured on event 000001000 from the training set – the score is usually 0.015 below the final leaderboard score):

Number of pairs	Score
1000	0.51
1600	0.56
5500	0.636
100000	0.73

The score plateaus after about 90000 pairs and doesn't increase further.

Machine Learning

Running the clustering algorithm a few times will produce similar, but slightly different solutions. Every one of these solutions has some good tracks that the other solution missed. Trying to merge these solutions in a naive way, by selecting the longest track for each hit, does not improve the score when the initial scores are good enough.

In our solution we built an algorithm to merge several solutions by using a Machine Learning (=ML) algorithm to evaluate the quality of the tracks. There were some discussions on taking a similar approach (4) (https://www.kaggle.com/c/trackml-particle-identification/discussion/58323), but we don't know of anyone who implemented it. As far as we know, most ML implementations where in an effort to construct or expand the tracks.

The nice thing about this kind of implementation is the fact that it is quite agnostic to the track construction algorithm. Hence it can be used to merge tracks from completely different algorithms.

General strategy

The steps are as follows:

- Produce different submission candidates sub_1, sub_2, ..., sub_N

- Create a machine learning model, which gives probabilities between 0 and 1 for each track candidate
- Merge two submission candidates by assigning to each hit the track, which has higher probability. Actually, we add to the probabilities a fraction of the track-length, and after a couple of merges, we also ask the new probability (from sub_j) to be at least 0.5 higher than the existing one
- Merge all submission candidates to get the final submission. The merging can be done sequentially, as we did in our final submission

Creating the machine learning model

The machine learning model we use is LightGBM. We choose 13 features per track:

- variance of x,y,z (these are the most important)
- minimum of x,y,z
- maximum of x,y,z
- mean of z
- volume_id of first hit
- number of clusters per track (i.e. are there many hits, which are close together?)
- number of hits divided by number of clusters

Training and validation

Training data from roughly 250 training events with roughly 5 million tracks were used:

- Correct tracks (target=1): All true tracks, taken from truth files
- Wrong tracks (target=0): These were generated by first running the clustering algorithm on an event, and then picking all generated tracks, whose hits belonged to at least two different particles (particle_id)

Good results were already achieved with 50000 training tracks, with diminishing gains after that. Training was a matter of minutes (some parameters we used: LightGBM with 3000 steps, learning_rate=0.05, 128 leafs). We also tested giving a "purity score" to each track (using "objective=xentropy" instead of "binary" in LightGBM), depending on how many percent of the suggested track belongs to the same particle. This did not change things enough to be further considered.

We tuned the hyperparameters of the model, based on the 3 training events with the ids 0, 1 and 2, which formed our validation set.

On the validation set, we achieved strong rates of roughly 95% for precision, accuracy and recall. However, these rates did not translate into big gains for the final score. One reason may be that the used correct/wrong tracks of our training/validation data did not resemble the final situation well enough, at which the model was employed. One step to improve this, was to take the "purity score" as described above, but this did not help our implementation.

Note: For our Machine Learning model to be helpful, it needs to distinguish correct from wrong tracks with very high

$$precision = TruePositives/(FalsePositives + TruePositives).$$

Also, it needs to do so for various sets of track candidates, especially those, which are generated if one tries to find tracks which originate far away from the origin. In these latter situations, often a lot of bad candidates are produced.

Approaches, which did not yield benefits

We tried dozens of other features (e.g. number of different volumes crossed, means of x,y), but got only negligible gains, probably because these features are closely related to the features we already use.

We also tried a deep neural network with a few hidden layers (and embeddings) and an LSTM architecture (where the input is a sequence of one value for each of the up to 20 hits of a track; the value was e.g. the volume-layer-module id). The first model gave similar performance to LightGBM, while the described LSTM model gave slightly worse results, though still better than we expected by just using the volume-layer-module id.

Finally, we also tried a scikit random forest implementation and Logistic Regression, but both gave worse results.

Results

In our final solution, we used the algorithm to sequentially merge 7 clustering solutions (mostly from previously generated submissions). We were able to increase our score by about 0.01. After the completion of the competition, we did another test where we merge 64 fast clustering solutions, using 1000 $(kt, z0)$ pairs each (i.e. a total of 64000 pairs). The score we got after expanding, for train event 1000, was 0.782. We get a similar score, but with longer runtime, by our usual algorithm of clustering 100000 pairs and then expanding.

Track Extension

The major drawback of the binning clustering algorithm is its sensitivity. The bins' sharp borders can leave a hit out of the track, although it is very close to the other hits. To overcome this issue, we employ a track expanding algorithm as a final step in our solution.

We start with a solution, which comes from merging different clustering solutions with our ML model. First, the track extension algorithm tries to improve the $(kt, z0)$ pair for each track. It does this by searching for a pair which will minimize the standard deviation of the track hits' features.

Using these refined $(kt, z0)$ values, the hits which are closest to a track are added to it. The distance between a hit and a track is measured by the difference between its features and the mean of the tracks' features.

We also tested a variant of this algorithm which measures the minimal distance to just one of the tracks' hits (compare nearest neighbor algorithm). This variant performed slightly better but was much slower.

The improvement gained by expanding depends on the score after the previous stages. We observed the following score improvements by using our track extension algorithm:

- 0.63 increases to 0.73,
- 0.73 increases to 0.79,
- and in our final run, 0.78 increased to 0.804

Finally we note, that expanding multiple times usually decreases the score.

Outlook

1. Our solution can be used in full, after being (easily) optimized for speed
    - The clustering algorithm can be improved as described above. Further improvements can be achieved by massively parallelizing the feature calculation using a GPU (as we explained above, every hit is treated separately, which makes the algorithm very suitable for parallel processing).
    - Both, the ML and the expanding algorithm where not optimized for speed, as they weren't the bottleneck of the solution. As an example, the expanding algorithm recalculates the features for all the

loose hits every time. Calculating the features only for hits with z>0 while expanding tracks with z>0 will immediately cut the running time of the algorithm in half.

2. Elements of our approach may be beneficial for any final solution
    - In particular, our binning algorithm creates a lot of good candidate tracks in a short period of time. It may be helpful to employ this at the beginning of a solution, in order to very quickly (<1 minute) detect 50% of all tracks (use small bins). One can then continue with running a different algorithm on the remaining hits.
    - Similarly, the machine learning algorithm, as well as the employed parameters and track extension algorithm may improve any final result, while just adding minutes (or less, after optimization) to the total runtime.
3. In our algorithm we didn't do any adjustments for the uneven magnetic field. If we incorporate @CPMP's findings here (3) (https://www.kaggle.com/c/trackml-particle-identification/discussion/63250#latest-372564), then our score improves immediately by 0.02 to about 0.82.
4. An obvious short-coming of our main approach is, that it is not suited to find tracks, which originate far from the origin. We did try to adjust our algorithm to work also in that situation, but had minimal success so far.

References

(1) @Grzegorz Sionkowski comment in "*how to score a track is good or not*" https://www.kaggle.com/c/trackml-particle-identification/discussion/60638#354053 (https://www.kaggle.com/c/trackml-particle-identification/discussion/60638#354053)

(2) "*Criteria for good features*" (https://www.kaggle.com/c/trackml-particle-identification/discussion/61590) (https://www.kaggle.com/c/trackml-particle-identification/discussion/61590)

(3) @CPMP "*Solution #9*" https://www.kaggle.com/c/trackml-particle-identification/discussion/63250#latest-372564 (https://www.kaggle.com/c/trackml-particle-identification/discussion/63250#latest-372564)

(4) @Heng CherKeng "*ensemble clustering?*" https://www.kaggle.com/c/trackml-particle-identification/discussion/58323 (https://www.kaggle.com/c/trackml-particle-identification/discussion/58323)

Chapter 2: Running full pipeline for train event 1000

Introduction

We will demonstrate how to get a score of about 0.75 for train event 1000, by using our aforementioned method. The steps are:

- Create 2 initial solutions: 2x 5500 pairs of $(kt, z0)$ for binning (in our original solution, we use 100000 pairs)
- Merge the 2 solutions using our machine learning algorithm (in our original solution, we use 7-8)
- Extend the tracks (In our original solution, we do this twice. At the first run, we extend hits according to whether their features are close enough to at least 1 hit, instead of the mean of the tracks' features.)

We first import necessary packages and load train event 1000, which will be the event we will be working on:

In [1]:

```
from IPython.display import HTML
import numpy as np
import sys
sys.path.insert(0, 'other/')
import pandas as pd
import datetime
import os
from ipywidgets import FloatProgress,FloatText
from IPython.display import display
import time
import pdb
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import cm
import gc
import cProfile
from tqdm import tqdm_notebook
%matplotlib inline
#make wider graphs
sns.set(rc={'figure.figsize':(12,5)})
plt.figure(figsize=(12,5))
path='files/'

from functions.other import calc_features, get_event, score_event_fast, load_obj
from functions.expand import *
from functions.cluster import *
from functions.ml_model import merge_with_probabilities,precision_and_recall,get

# auto load changed modules:
%load_ext autoreload
%autoreload 2

event_num=0
event_prefix = 'event00000100{}'.format(event_num)
hits, cells, particles, truth = get_event(path,event_prefix)
```

```
<Figure size 864x360 with 0 Axes>
```

Clustering

Define parameters and run clustering, twice:

In [2]:

```
history=[]
weights={'pi':1,'theta':0.15}
stds={'z0':7.5, 'kt':7.5e-4}
d =    {'sint':[225,110,110,110,110,110],
        'cost':[225,110,110,110,110,110],
          'phi':[550,260,260,260,260,260],
        'min_group':[11,11,10,9,8,7],
        'npoints':[500,2000,1000,1000,500,500]}
filters=pd.DataFrame(d)
nu=500

resa1=clustering(hits,stds,filters,phik=3.3,nu=nu,truth=truth,history=history)
resa1["event_id"]=event_num
score = score_event_fast(truth, resa1.rename(index=str, columns={"label": "tracl
print("Your score: ", score)

resa2=clustering(hits,stds,filters,phik=3.3,nu=nu,truth=truth,history=history)
resa2["event_id"]=event_num
score = score_event_fast(truth, resa2.rename(index=str, columns={"label": "tracl
print("Your score: ", score)
```

s rate: 0,965622907864698

add score: 0,06533484379

Rest size: 59600

Group size: 7

filter: 260

```
100%|██████████| 5500/5500 [03:48<00:00, 24.11it/s]

took 228.19628 sec
Your score:  0.63804680154
```

Employ Machine Learning

We have prepared a smaller sized training and test set. We load it directly from a pkl-file (compare "Create training.ipynb"). In particular, the below training data contains roughly 140k tracks, instead of the 5 million tracks, which we used to train the model of our original solution.

In [8]:

```
df_train=load_obj('files/df_train_v2-reduced.pkl')
df_test=load_obj('files/df_test_v1.pkl')
y_train=df_train.target.values
y_test=df_test.target.values
print("The dataframe with all features:")
print(df_train.shape,df_test.shape)
display(df_train.head())
print("Features/data for each track:",df_train.columns.values)
```

```
The dataframe with all features:
(138645, 15) (35636, 15)
```

	nclusters	nhitspercluster	svolume	target	xmax	xmin	xvar	yma
35636	5	1.000000	8	0	232.5970	65.19430	5382.914534	451.9450
35637	5	1.200000	8	0	-161.7980	-284.92200	2356.988817	228.1580
35638	6	1.000000	8	0	447.0080	4.62093	25546.189171	-32.2145
35639	10	1.200000	8	1	83.0611	1.76645	740.710959	404.8060
35640	7	1.285714	8	1	-33.0381	-465.40300	19380.462489	-2.2652

```
Features/data for each track: ['nclusters' 'nhitspercluster' 'svolume'
'target' 'xmax' 'xmin' 'xvar'
 'ymax' 'ymin' 'yvar' 'zmax' 'zmean' 'zmin' 'zvar' 'event_id']
```

We now proceed to create and train the LightGBM model:

In [9]:

```
import lightgbm
s=time.time()
columns=['svolume','nclusters', 'nhitspercluster', 'xmax','ymax','zmax', 'xmin',
rounds=1000
round_early_stop=100
parameters = { 'subsample_for_bin':800, 'max_bin': 512, 'num_threads':8,
               'application': 'binary','objective': 'binary','metric': 'auc','bo
               'num_leaves': 128,'feature_fraction': 0.7,'learning_rate': 0.05,'
train_data = lightgbm.Dataset(df_train[columns].values, label=y_train)
test_data = lightgbm.Dataset(df_test[columns].values, label=y_test)
model = lightgbm.train(parameters,train_data,valid_sets=test_data,num_boost_roun
print('took',time.time()-s,'seconds')
```

```
Training until validation scores don't improve for 100 rounds.
[200]   valid_0's auc: 0.97131
[400]   valid_0's auc: 0.973004
[600]   valid_0's auc: 0.973358
Early stopping, best iteration is:
[685]   valid_0's auc: 0.973458
took 5.223278522491455 seconds
```

Judge machine learning model

We doublecheck the model's performance, by calculating its precision, recall and accuracy on the validation set:

In [10]:

```
y_test_pred=model.predict(df_test[columns].values)
precision, recall, accuracy=precision_and_recall(y_test, y_test_pred,threshold=(
precision, recall, accuracy=precision_and_recall(y_test, y_test_pred,threshold=(
precision, recall, accuracy=precision_and_recall(y_test, y_test_pred,threshold=(
```

```
Threshold 0.1  --- Precision: 0.8028, Recall: 0.9903, Accuracy: 0.8693
Threshold 0.5  --- Precision: 0.9115, Recall: 0.9259, Accuracy: 0.9152
Threshold 0.9  --- Precision: 0.9711, Recall: 0.7144, Accuracy: 0.8414
```

Use machine learning model

Merge the two submissions, which were generated using clustering, based on the probabilities of its track candidates:

In [11]:

```
preds={}
preds[1]=get_predictions(resa1,hits,model)
preds[2]=get_predictions(resa2,hits,model)
print('Merge submission 0 and 1 into sub01:')
sub01=merge_with_probabilities(resa1,resa2,preds[1],preds[2],None,length_factor=
score = score_event_fast(truth, sub01)
print('Score:',score)
```

```
Merge submission 0 and 1 into sub01:
Score: 0.6773030530600002
```

Expand tracks

In [12]:

```
mstd_vol={7:0,8:0,9:0,12:2,13:1,14:2,16:3,17:2,18:3}
mstd_size=[4,4,4,4,3,3,3,2,2,2,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,(
weights={'theta':0.1, 'phi':1}
nresa=expand_tracks(sub01,hits,5,16,5,7,mstd=8,dstd=0.00085,phik=3.3,max_dtheta=
nresa['event_id']=0
score = score_event_fast(truth, nresa)
print("Your score: ", score)
```

```
100%|██████████| 100/100 [00:08<00:00, 12.14it/s]
```

calculating:

```
100%|██████████| 7305/7305 [03:49<00:00, 31.86it/s]

Your score:  0.7543045200999999
```

We achieved a final score of 0.754.

Appendix: Create 4*2250 pairs clustering, merge with ML, and expand

In [14]:

```
def binary_cluster_merge(stage):
    if stage==0:
        weights={'pi':1,'theta':0.15}
        stds={'z0':7.5, 'kt':7.5e-4}
        d =    {'sint':[225,110,110],
                'cost':[225,110,110],
                  'phi':[550,260,260],
                'min_group':[11,11,10],
                'npoints':[250,1250,750]}
        filters=pd.DataFrame(d)
        nu=250
        res=clustering(hits,stds,filters,phik=3.3,nu=nu,truth=truth,history=hist
        res["event_id"]=event_num
        score = score_event_fast(truth, res.rename(index=str, columns={"label":
        print("Your score: ", score)
    else:
        res1=binary_cluster_merge(stage-1)
        res2=binary_cluster_merge(stage-1)
        preds1=get_predictions(res1,hits,model)
        preds2=get_predictions(res2,hits,model)
        print('Merge submission stage:',stage)
        res=merge_with_probabilities(res1,res2,preds1,preds2,None,length_factor=
        score = score_event_fast(truth, res)
        print('Score:',score)
    return(res)

res = binary_cluster_merge(2)
mstd_vol={7:0,8:0,9:0,12:2,13:1,14:2,16:3,17:2,18:3}
mstd_size=[4,4,4,4,3,3,3,2,2,2,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,(
weights={'theta':0.1, 'phi':1}
res=expand_tracks(res,hits,5,16,5,7,mstd=8,dstd=0.00085,phik=3.3,max_dtheta=0.9
res['event_id']=0
score = score_event_fast(truth, res)
print("Your score: ", score)
```

```
100%|██████████| 2250/2250 [01:41<00:00, 22.20it/s]

took 101.40930 sec
Your score:  0.57850939805
Merge submission stage: 1
Score: 0.6286514243800001
Merge submission stage: 2

  0%|          | 0/100 [00:00<?, ?it/s]

Score: 0.6753546458300002

100%|██████████| 100/100 [00:08<00:00, 12.05it/s]

calculating:

100%|██████████| 7186/7186 [03:47<00:00, 31.62it/s]

Your score:  0.7530926931899999
```

This yielded a final score of 0.753.