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.

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 paralleling on 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(NlogN)). 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, this method is limited because 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 Kaggle, 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 choosing the 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, using 100000 loop iterations each, 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 hits with close feature together. The first stage in clustering is finding features which would describe a track and which we can calculate separately for every 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 may 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 '|z0|<5.5mm').

This kind of helix can be defined by 3 numbers, its Radius, its tangential angle at the origin in the xy plan (=the direction of the particle when it is created, in xy plan), the slop of the helix (how fast the particle moves in the Z direction compared to its velocity in xy).

We will define:

R – helix radius

θ – tangential angle in the xy plan

ϕ – Slope

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

where px, py, pz, are the particle's initial momentum

If we take a family of helixes with a radius R, we can calculate for every hit '(x,y,z)' the values for theta and phi. We define

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

We use kt>0 for particle rotating to the CW and kt<0 for CCW rotation

Using sum trigonometry, we will get:

$$\theta\_ = \arctan \frac{py}{px}$$

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

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

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

(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.

If $\theta_1 = \phi + \epsilon$ and $\theta_1 = \phi - \epsilon$, $\theta_1 - \theta_2 = 2\epsilon$ on a circle, but if we calculate $\theta_1 - \theta_2$ we will get $2\pi + 2\epsilon$. To solve this issue we will use 2 features instead:

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

$\phi\_$ 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)).

Sparse Binning Clustering

Binning is the easiest way to cluster, in this method we divide the values of each feature to constant areas and a cluster is formed by all the hits whos features are all in the same area.

As an example, if we have one feature in the range `[-1,1]` we can decide to divide it to 20 equal bins of size `0.1: [-1,-0.9), [-0.9,0.8), …,[0.9,1]`. We will assign every bin a number 0 , …, 19 and define the hit's feature with this value. If we have 3 features for a hit we will define that hit with the 3 values of bins. As an example:

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

Two hits which have the exact 3 values, are clustered together to form track candidate.

In general, the bins don't have to spread uniformly, and every feature can have a different number of bins.

In Sparse Binning we use a very simple binning algorithm. We start by using uniform spread bins which can be easily calculated:

For feature `F1` in the range `[-1,1]` we define `B1 = int(F1*k1)` where the number of bins is `2*k1`

i.e. we multiply the feature by half the number of bins and take the integer part.

To combine few features together and get the cluster id (track_id) we just offset each feature by multiplying it by a number which is big enough.

If we have 2 features F1, F2 and we want to have 2*k1, 2*k2 bins for each feature, we can define:

```
(a) Track_id = int(F1*k1) + (2*k1+1)*int(F2*k2)
```

And that's it clustering is finished!!

This calculation takes about 4mSec for 120,000 hits on 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 boarders 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 (1), Hit1, Hit2 will never cluster together no matter how many bins we'll have.

This issue is solved by adding a random number in the range `[0,1)` to every feature in the binning calculation, making (a):

```
(b) Track_id = int(F1*k1+r1) + (2*k1+1)*int(F2*k2+r2)
```

Where r1, r2 are random numbers with uniform distribution in the range `[0,1)`. r1, r2 are changed every time we cluster.

When compared to dbscan, sparse clustering has one big disadvantage, it's sensitivity, it can miss some of

Simple Merging

Every time we cluster with a set of kt, z0, every hit gets a new track_id, to have a full solution we want to merge two lists of track id's together. The most efficient way we found was 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_id's 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, hit#1 will choose 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 Kaggle kernel platform. The numpy.unique computational complexity is O(NlogN) because it sorts the values. While if we have enough memory, this task can easily be done in O(N). (We can also reduce the amount of memory needed by hashing).

In our main loop we measure the length of a track twice, the first time we do as we described above to create a track candidate and then we measure again to make sure the track is still long enough as hits may decide to choose another track in the first round.

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

Clustering – the full algorithm

The algorithm iterate over random selected pairs – (z0,kt). 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. Tracks are examined for hits coming from the same detector (equal Volume_id, Layer_id, and Module_id) or from the same layer. There can't be two hits from the same detector or 3 from the same layer. The extra hits, are discovered by measuring the distance between the hit and the center of gravity of the track. These hits are removed from the track.

In this step the algorithm also permanently set tracks which are long enough (hits won't we taken away or added to this track any more).

While running the algorithm can change the number of bins per feature and the length of the minimal track to set, these settings are user defined parameters.

Clustering results:

The score common score after the clustering algorithm with as a function of the number of (z0,kt) pairs used is presented in the following table (measured on event 000001000 from the training set – usually 0.015 below final LB score):

Pairs	Score
1000	0.51
1600	0.56
5500	0.636
100000	0.73

The score plateau after about 90000 pairs and wouldn’t get higher

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 marge these solutions in a naïve 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 few solutions by using a Machine Learning (=ML) algorithm to evaluate the quality of the tracks. Although there were some discussions on taking a similar approach (4) (https://www.kaggle.com/c/trackml-particle-identification/discussion/58323), 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 general strategy for this solution is 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
- 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 chose 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 We tried many more features (e.g. number of different volumes crossed, means of x,y etc.), but we saw only negligible gains, probably because these features are closely related to the features we already use.

Training and validation

The model was trained using 250 events from the training set. The true tracks (target=1) were selected using the truth file. The wrong tracks (target=0) 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).

Validation was done using the 3 training events with the ids 0, 1 and 2.

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 out score by about 0.01. After the completion of the competition, we did another test where we merge 64 fast clustering solutions, using 1000 (z0,kt) 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 expansion, which is slower.

Track Extension

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

The final clustering and merging solution it taken as a basis for this step.

First, the algorithm tries to improve the (z0,kt) pair for each track. It does this by searching for a pair which will minimize the standard deviation of the track hits.

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

We also tested a variant of this algorithm which measure the minimal distance to one of the track's hit (as in 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.

0.63 score can jump to 0.73,

0.73 would go to 0.79

and in our final run 0.78 was improved to 0.804

This input to this stage must come from good clustering algorithm and cascaded extending will usually degrade the score.

Outlook

1. As a full solution our approach can be easily optimized for speed:
    - The clustering algorithm can be improved as described above. Farther improvement for real world application can be achieved by massively paralleling 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 at all 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 by half.
2. Elements of our approach may be beneficial for any final solution to this problem.
    - 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 uneven magnetic field. If we incorporate @CPMP's findings here (3) (https://www.kaggle.com/c/trackml-particle-identification/discussion/63250#latest-372564). The score of the algorithm improves immediately by 0.02 to about 0.82.
4. An obvious short-coming of 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 0.755 for train event 1000, by using our aforementioned method. The steps are:

- Create 2 initial solutions: 2x use 5.500 pairs of (kt,z0) for binning (in our original solution, we use 100.000 pairs)
- Merge the 2 solutions using a machine learning algorithm (in our original solution, we use 8)
- Extend the tracks (in our original solution, we run this twice, while at the first time, we extend hits according to whether their values for (1) and (2) are close enough to at least 1 hit, instead of the mean of the track)

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_featur
es,get_predictions

# the following two lines are for changing imported functions, and not needing to resta
rt kernel to use their updated version
%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": "track_id"}))
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": "track_id"}))
print("Your score: ", score)
```

```
100%|██████████████████████████████████████████████████████████████████
█████████| 5500/5500 [05:51<00:00, 15.64it/s]

took 352.03976 sec
Your score:  0.6364404584700001
```

```
100%|██████████████████████████████████████████████████████████████████
█████████| 5500/5500 [04:44<00:00, 19.32it/s]

took 284.88436 sec
Your score:  0.6376050015900001
```

Employ Machine Learning

[Note: For our final solution, the methods described in this chapter gave around +0.01 to the LB score. Certain benefits from these methods have already been captured by the function, which expands the tracks.]

We have prepared the training and test data and load it directly from a pkl-file:

In [9]:

```
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:")
display(df_train.head())
print("Features for each track:",df_train.columns.values)
```

```
The dataframe with all features:
```

	nclusters	nhitspercluster	svolume	target	xmax	xmin	xva
35636	5	1.000000	8	0	232.5970	65.19430	5382.914534
35637	5	1.200000	8	0	-161.7980	-284.92200	2356.988817
35638	6	1.000000	8	0	447.0080	4.62093	25546.18917
35639	10	1.200000	8	1	83.0611	1.76645	740.710959
35640	7	1.285714	8	1	-33.0381	-465.40300	19380.46248

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

In the competition, we used roughly 250 events for training, but the additional improvement to just using 13 events is not too big. We now create the LightGBM model, using the mentioned training data and features:

In [10]:

```
import lightgbm
s=time.time()
# choose which features of the tracks we want to use:
columns=['svolume','nclusters', 'nhitspercluster', 'xmax','ymax','zmax', 'xmin','ymin',
'zmin', 'zmean', 'xvar','yvar','zvar']
rounds=1000
round_early_stop=50
parameters = { 'subsample_for_bin':800, 'max_bin': 512, 'num_threads':8,
               'application': 'binary','objective': 'binary','metric': 'auc','boosting'
: 'gbdt',
               'num_leaves': 128,'feature_fraction': 0.7,'learning_rate': 0.05,'verbos
e': 0}
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_round=round
s,early_stopping_rounds=round_early_stop,verbose_eval=50)
print('took',time.time()-s,'seconds')
```

```
Training until validation scores don't improve for 50 rounds.
[50]    valid_0's auc: 0.957703
[100]   valid_0's auc: 0.965927
[150]   valid_0's auc: 0.96966
[200]   valid_0's auc: 0.971289
[250]   valid_0's auc: 0.972094
[300]   valid_0's auc: 0.972588
[350]   valid_0's auc: 0.972787
[400]   valid_0's auc: 0.972987
[450]   valid_0's auc: 0.973106
[500]   valid_0's auc: 0.973178
[550]   valid_0's auc: 0.973262
[600]   valid_0's auc: 0.973249
Early stopping, best iteration is:
[563]   valid_0's auc: 0.973293
took 7.472992658615112 seconds
```

Judge machine learning model

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

[Note: For ML to be helpful in our situation, it needs to distinguish correct from wrong tracks with very high precision=true_positives/(false_positives+true_positives)). Also, it needs to do so for various sets of track candidates (especially such, which are generated if one tries to find tracks which originate far away from the origin; in those situations, often a lot of bad candidates are produced).]

In [11]:

```
y_test_pred=model.predict(df_test[columns].values)
precision, recall, accuracy=precision_and_recall(y_test, y_test_pred,threshold=0.1)
precision, recall, accuracy=precision_and_recall(y_test, y_test_pred,threshold=0.5)
precision, recall, accuracy=precision_and_recall(y_test, y_test_pred,threshold=0.9)
```

```
Threshold 0.1  --- Precision: 0.7974, Recall: 0.9910, Accuracy: 0.8653
Threshold 0.5  --- Precision: 0.9119, Recall: 0.9257, Accuracy: 0.9154
Threshold 0.9  --- Precision: 0.9711, Recall: 0.7055, Accuracy: 0.8370
```

Use machine learning model

Calculate the probabilities for the tracks in those two submissions (small optimization: take also length of track into account, and after a couple of merged submissions, ask the probability of the track from the new submission to be at least C higher than the current probability; this latter option is not used in this kernel, but was used when merging >= 4 submissions).

Then merge both submissions, based on the probabilities of its track candidates:

In [12]:

```
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=0.5)
score = score_event_fast(truth, sub01)
print('Score:',score)
```

```
Merge submission 0 and 1 into sub01:
Score: 0.6765421439
```

Expand tracks

In [7]:

```
mstd_vol={7:0,8:0,9:0,12:2,13:1,14:2,16:3,17:2,18:3}
mstd_size=[4,4,4,4,3,3,3,2,2,2,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
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=0.9*np.
pi/2,mstd_vol=mstd_vol,mstd_size=mstd_size,weights=weights,nhipo=100)
nresa['event_id']=0
score = score_event_fast(truth, nresa)
print("Your score: ", score)
```

```
100%|██████████████████████████████████████████████████████████████████████
███████████| 100/100 [00:19<00:00,  5.12it/s]

100%|██████████████████████████████████████████████████████████████████████
█████████| 7286/7286 [04:13<00:00, 28.76it/s]

Your score:  0.7557564998499999
```

Binary merge 4*2250pairs clustering solution.

In [8]:

```
def binary_cluster_merege(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=history)
        res["event_id"]=event_num
        score = score_event_fast(truth, res.rename(index=str, columns={"label": "track_
id"}))
        print("Your score: ", score)
    else:
        res1=binary_cluster_merege(stage-1)
        res2=binary_cluster_merege(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=0.5)
        score = score_event_fast(truth, res)
        print('Score:',score)
    return(res)

res = binary_cluster_merege(2)
mstd_vol={7:0,8:0,9:0,12:2,13:1,14:2,16:3,17:2,18:3}
mstd_size=[4,4,4,4,3,3,3,2,2,2,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
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*np.pi/2
,mstd_vol=mstd_vol,mstd_size=mstd_size,weights=weights,nhipo=100)
res['event_id']=0
score = score_event_fast(truth, res)
print("Your score: ", score)
```

```
100%|██████████| 2250/2250 [02:09<00:00, 17.31it/s]

took 130.12909 sec
Your score:  0.57609589578
```

```
100%|██████████| 2250/2250 [02:20<00:00, 16.01it/s]

took 140.68634 sec
Your score:  0.57515281957
Merge submission stage: 1
Score: 0.62738866609
```

```
100%|██████████| 2250/2250 [02:07<00:00, 17.59it/s]

took 128.04464 sec
Your score:  0.5806932663500001
```

```
100%|██████████| 2250/2250 [02:07<00:00, 17.71it/s]

took 127.18123 sec
Your score:  0.5761987493699999
Merge submission stage: 1
Score: 0.62984464579
Merge submission stage: 2
Score: 0.67136842166

100%|██████████| 100/100 [00:19<00:00,  5.18it/s]

100%|██████████| 7181/7181 [04:10<00:00, 28.66it/s]

Your score:  0.7517000975400001
```