Pushing DBSCAN To Its Limit

Jean-François Puget
IBM France, Les Taissounières, 06560 Valbonne
j-f.puget 'at' fr 'dot' ibm 'dot' com

Competition Name :	TrackML
Team Name :	DBSCAN forever
Private Leaderboard Score:	0.80114
Public Leaderboard Score:	0.80225

Summary

The approach used here is unsupervised machine learning (clustering) in an iterative approach. It only looks for tracks that originate from the vertex (origin close to z axis and values of z between -50 and 50). At each iteration, hit coordinates are transformed so that hits belonging to tracks with a given radius and z at origin are close to each other after the transformation. The transformation includes some systematic deviation from perfect helix at high values of z. Then DBSCAN is used to find clusters in the transformed space. These clusters are new candidate tracks that are merged with the candidate tracks found by previous iterations. Hit belonging to more than one candidate tracks are assigned to the candidate track that maximizes some track quality measure.
Three variants of this approach have been run, they differ in the way hit coordinates are transformed or the volumes considered. By merging the track candidates from the three models we get private/public LB scores of *0.80114 / 0.80225* which ranks 9^{th} on the leaderboard.
When cleaning the code for publication, I fixed one bug and reran the corrected code. With only two models instead of three the fixed code yields private/public LB scores of *0.80047 / 0.80089*. This document described the fixed code as it is both simpler than the one used in the competition and equivalent for LB score and ranking.

High-level description

The approach is very simple, the code for building one model is less than 200 lines of Python. It is rather slow, but the code can be made much faster. Running speed was not part of the evaluation, and I preferred to work on improving accuracy of the method than speeding it up. Moreover, one can easily control the tradeoff between speed and accuracy using the number of iterations. A score above 0.51 can be obtained in 2 minute 30 seconds per event with one cpu core.

The approach is also quite different from all the approaches shared by top teams in the forum because it uses neither supervised learning, nor track fitting. Some teams also used DBSCAN, but they complemented it with track fitting or track extension, or with supervised learning methods.

The algorithmic outline of the approach is as follows.

1. Prepare data using train events ranging from 1010 to 1099 included.
    a. Compute track parameters for each track that originates from the vertex in these events.
    b. Assign a unique *vl_id* to each <*volume_id*, *layer_id*> pair. Then sort each event hit data by *track_id* then *z*. Then compute the frequency of each subsequence of 4 *vl_id* from all tracks. We call these subsequences *tracklets*.
2. For each test event
    a. Assign each hit to track candidate 0
    b. For a given number of iterations:
        i. Select randomly track parameters from those computed in step 1.a.
        ii. Compute transformed hit coordinates using the selected track parameters.
        iii. Run DBSCAN. Its output clusters are tracks candidates if they contain between 2 and 20 hits.
        iv. Compute number of volumes and quality of each candidate track. Quality is computed using the frequencies of tracklets computed in 1.b.
        v. Assign hits to new track candidates if the number of volumes and the quality of the next track candidate is better than the current track of the hit. Track *0* is considered to have *0* volumes and the lowest possible quality
    c. Save the tracks as for each event as *final*
3. Repeat step 2 by only keeping hits from volumes 7,8, and 9. Save the tracks for each event as *final_inner*
4. Merge tracks from *final* and *final_inner*
    a. tracks from both models that overlap more than a given threshold are merged. Remaining hits are assigned to tracks from *final* if possible. Remaining hits are assigned to tracks from *final_inner* if possible.
5. Submit the result

Main parameters to tune were the number of iterations, and the *eps* parameter for DBSCAN. These were tuned on train events ranging from 1000 to 1003. The average score for these four events was always very close to the public leaderboard score, with a difference smaller than 0.001 in general.

Scientific Details

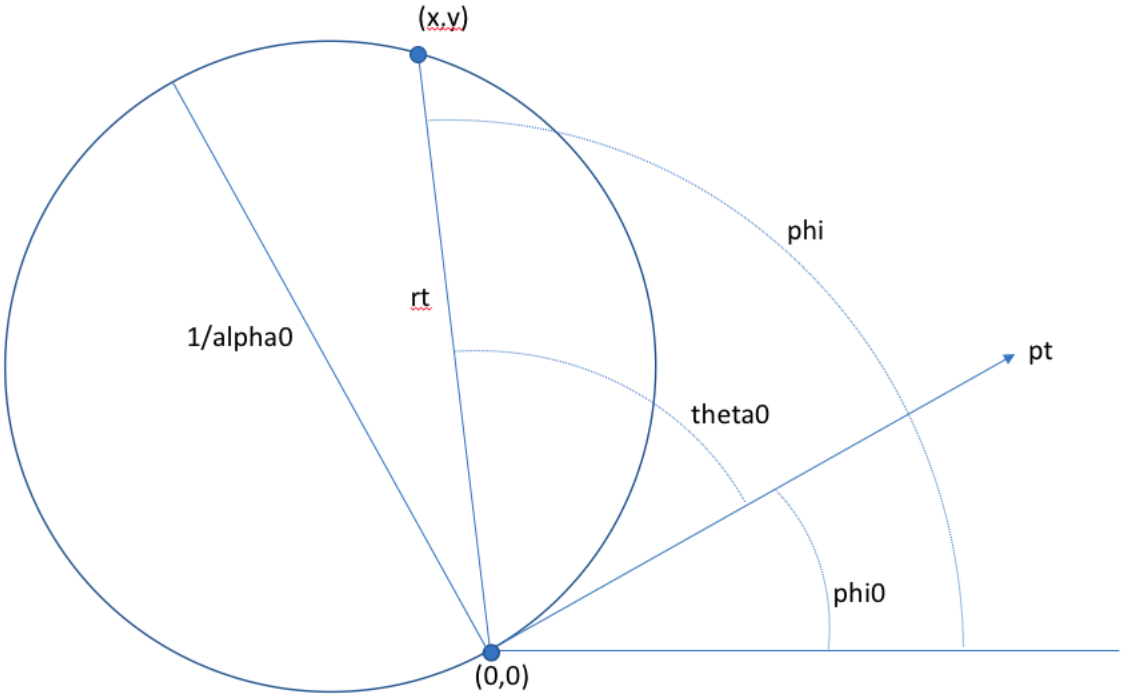
An helix that starts from z axis can be described by 4 parameters:

- z at origin *z0*
- diameter *d0* or, equivalently, inverse of diameter *alpha0 = 1/d0*
- angle at origin in *(x,y)* plane *phi0*
- slope *zr*

The last 3 can be expressed as functions of the momentum at origin, *px*, *py*, *pz*, and *pt = sqrt(px² + py²):*

alpha0 = C / pt
phi0 = arctan2(py, px)
zr = pz/pt

Given the value of *C* was not provided, I computed *alpha0* from momentum at origin and first hit coordinates. I selected this way because particles have no interaction with matter until first hit, hence tracks follow a perfect helix. See Appendix A for details.



My code loops over <*z0*, *alpha0*> pairs. Rather than estimating a distribution I just sample from the tracks in 90 train events. For each pair I *'unroll the helix*' [1], i.e. compute *phi0* and *zr* as:

$$phi0 = phi +- theta0 * theta\_ratio$$
$$arc = theta0 / alpha0$$
$$zr = (z - z0) / arc$$

where *rt* is the distance to origin in transverse plane, *phi* the angle in transverse plane, and *theta0* is the unrolling angle (see picture above):

$$rt = sqrt(x^2 + y^2)$$
$$phi = arctan2(y, x)$$
$$theta0 = arcsin(rt * alpha0)$$

and where *theta_ratio* is a correction factor for some variation of the magnetic field. This factor is described below.

In one iteration I add *theta0 * theta_ratio* to *phi*, and in the next iteration I subtract it.

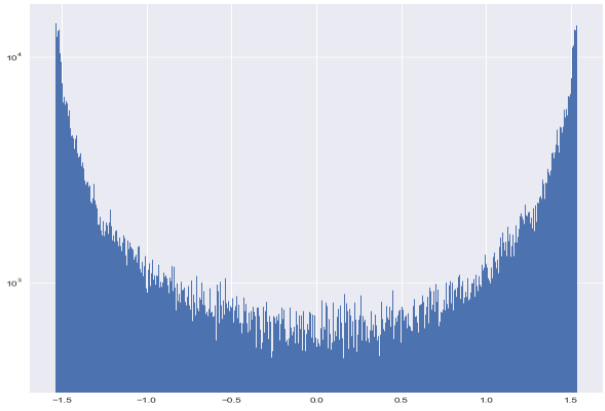
In order to cope with the discontinuity at *pi* and *-pi* I use *cos(phi0)* and *sin(phi0)* as features for DBSCAN.

The distribution of *zr* is highly skewed. Many have use *arctan* to unskew it, but I found that using *arcsinh* was way more effective. I actually use:

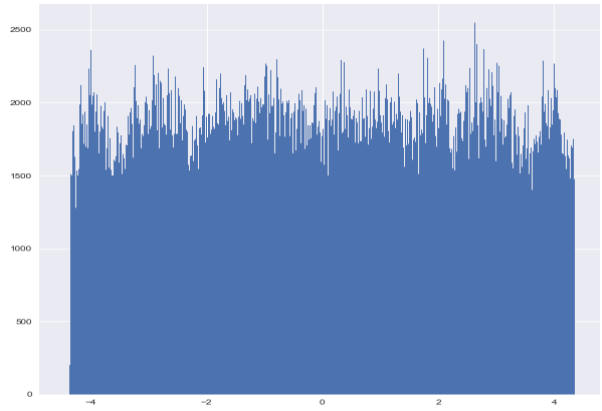
arcsinh(zr / 0.7) / 3.5

as the third feature for DBSCAN.

The picture below gives the distribution of *arctan(zr)*, and *arcsinh(zr)*. The latter is way more uniform.

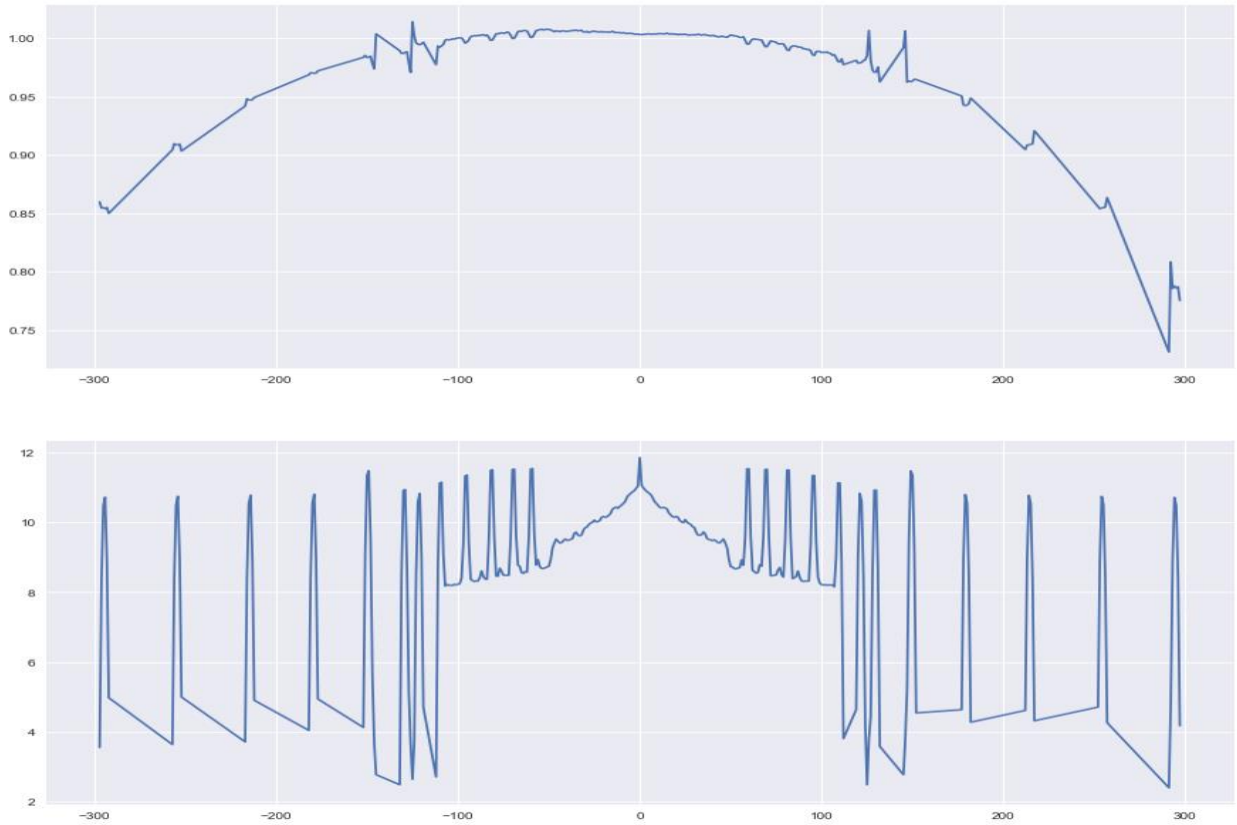


Arctan(zr) values



Arcsinh(zr/0.7) values

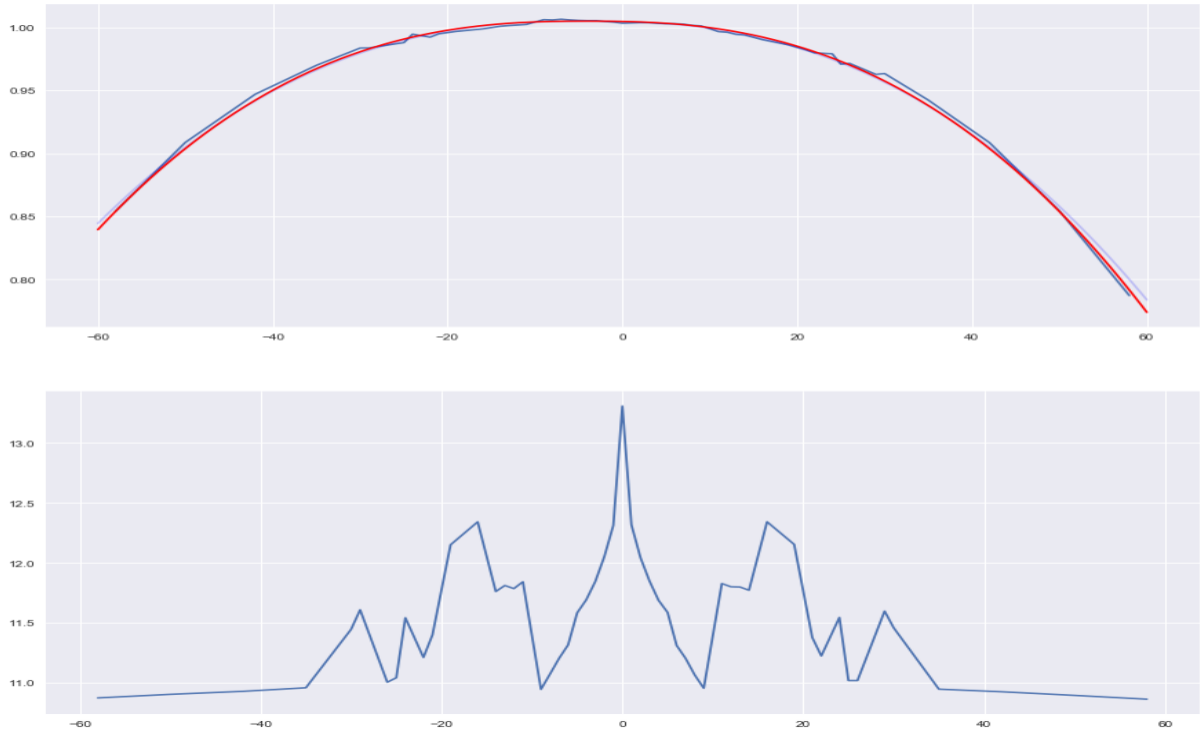
The last twist is to model the uneven magnetic field at high values of *z*. The picture below shows the relative difference between the theoretical angle, and the median of measured angle as a function of *z* scaled by *0.1*, the picture below gives the number of hits in log scale:



The best way I found was to multiply *theta0* with a correction that depends on *z* when computing *phi0*:

theta_ratio = 1.005 - (abs(z + 200) / 6000)$^{2.4}$

The picture below shows a smoothed average of the median measured angle deviation (in blue), and my correction function (in red), as function of *z* scaled by *0.02*:



We see that the deviation can be significant, up to more than 20%.

DBSCAN is run at each iteration on the transformed hits coordinates :

<cos(phi0), sin(phi0), arcsinh(zr / 0.7) / 3.5>

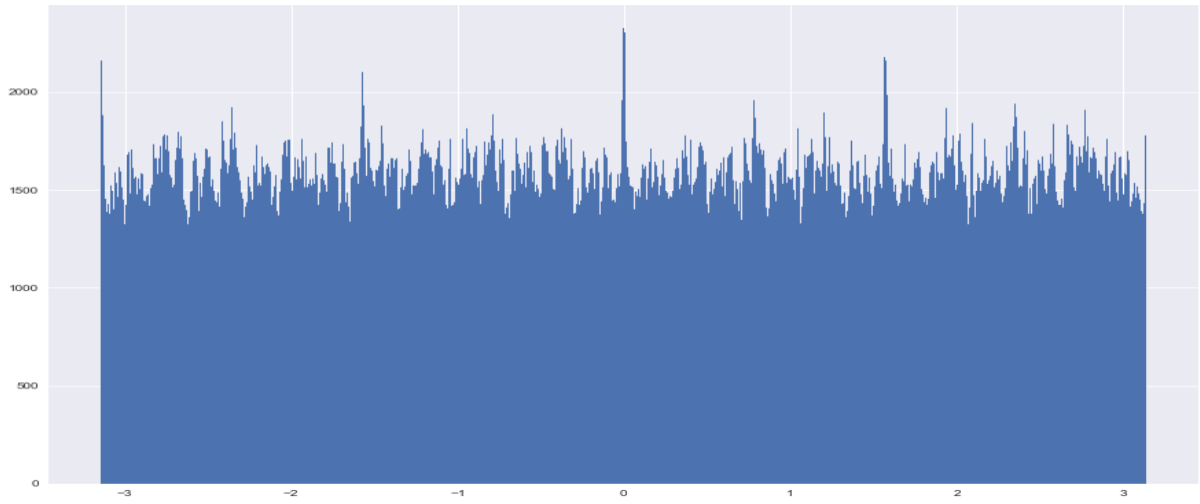
DBSCAN output is merged with existing tracks in a simple way: for each track, or candidate track, I compute the number of layers with hits from the track as well as the track quality. A hit is assigned to a new candidate track if the new candidate track has both more layers and a better quality than the current track of the hit.

The quality of each track candidate in test events is computed using the average of its tracklets frequencies multiplied by the number of layers of the candidate track. The track quality is very effective in removing tracks that do not make sense, for instance tracks that skip a layer entirely.

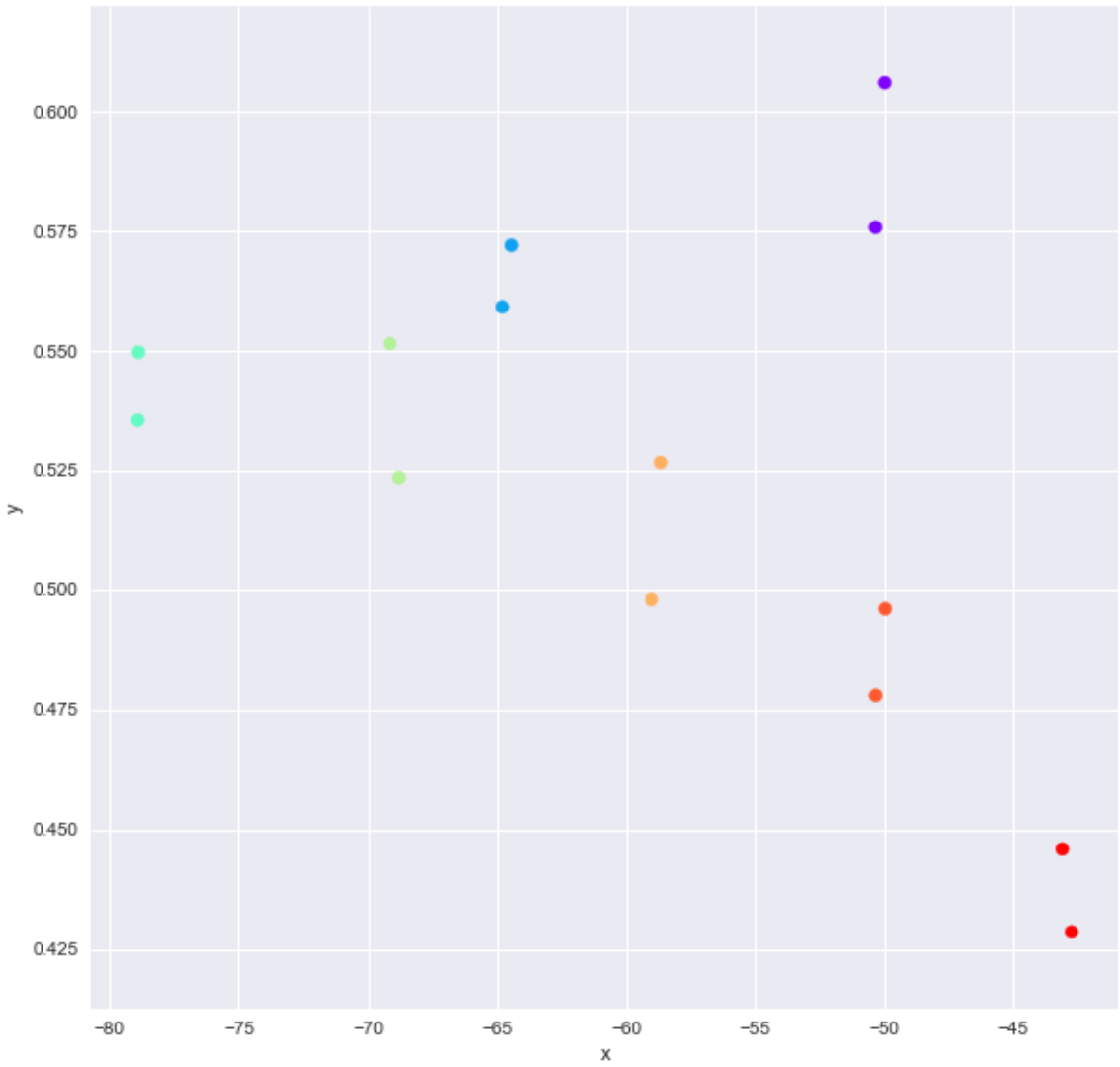
Interesting findings

What strikes me the most was the variation on the magnetic field. More precisely, the deviations between the theoretical angle predicted by the theory, and the actual angles, see the picture above. We see that the deviation can be more than 20%, which is very significant.

The second surprising fact to me is that the distribution of phi values seems to follow some pattern, with peaks at *pi / 2*^{k}, as shown by the picture below.



The last surprise was to find tracks that were clearly produced by some bug in the simulator. I reported it, and few others in the competition forum [2]. I was told that the impacted tracks were a very small number. Picture below shows one of the strangests I found.



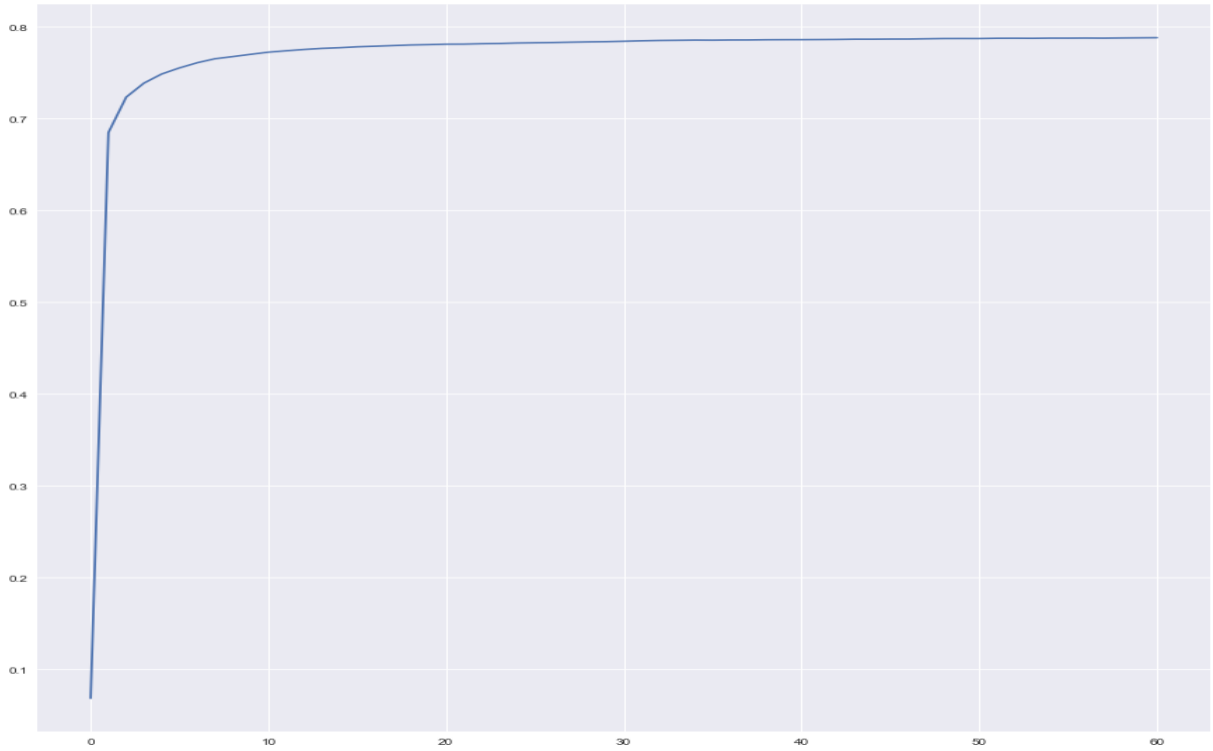
Weird track in (*x*, *y*) plane. Color is *z*.

Simple Features and Methods

The simplified version of my solution is to run the first model only. Computing it requires less than 200 lines of code.

When ran for 60000 iterations it yields private/public LB score of *0.78701 / 78772*

Running it for so many iterations is rather slow, about 10 hours on an Intel i7 core, but one can get very good result with much less iterations. The picture below shows the local score as a function of the number of iterations in thousands:



Running the same model for 150 iterations yields private/public scores *0.51202 / 0.51095*. This takes about 150 seconds per event. 0.7 is reached in about half an hour.

Model Training and Execution Time

There is no model training per se, but there are some statistics preparation that takes about 10 minutes on an i7 core. Memory usage if few GB. The code uses numpy and pandas from Anaconda distribution. These are optimized and yield speed of execution comparable to Intel Python distribution.

Model 1 takes about 1 hour for 4000 iterations, and Model 2 takes about 1 hour for 8000 iterations on an i7 core at 4.2 GHz. Merging tracks takes less than a minute. To get 0.8 on the leaderboard requires about 20 hours per event on an i7 core. A score of 0.7 can be obtained in about half an hour per event, and 0.5 can be obtained in 150 seconds per event.

Outlook

My approach is limited in various ways. I identified several ways to improve it, and started to work on the first one, but could not finish in time.

1. **Look for tracks that do not origin from the vertex.** I implemented an approach similar to the above for that. The main difference is the hit coordinate transformation. Tracks are described by the point closest to origin among points closest to z axis, the slope, and the diameter of the track. The angle correction *theta0* is a bit more convoluted, but the principle is the same as for previous models. This code was finding a fair number of tracks, but I did not find a good way to merge these tracks with the ones found by the two models above. I only could find a small upside of about 0.003. When using that with the two models described above I get private/public scores of *0.80378/0.80421* which would rank 8^{th}.
2. **Get better track merge**. Using number of volumes traversed and frequency of tracklets is very effective, but it may be possible to improve it by training a classifier to predict which of two candidate tracks a hit should be assigned to. This is related to what team #7 did [3.
3. **Find tracks beyond pi.** The approach is limited to *theta0* ranging from *-pi* to *pi*. Loopers can have tracks that extend beyond a half circle in the *(x,y)* plane. One way to catch these would be to extend tracks beyond half circle by fitting a circle to the track candidate. More generally, track extension methods as used by other teams could be applied as a postprocessing step here.
4. **Use direction information.** The cells data contain enough information to compute some direction information [4]. I wanted to use that to help DBSCAN and get more accurate clusters.

References

[1] *Unrolling the helix* by Konstantin Lopuhin https://www.kaggle.com/c/trackml-particle-identification/discussion/57180

[2] *Simulator bugs* by Jean-François Puget https://www.kaggle.com/c/trackml-particle-identification/discussion/60182

[3] *Team #7 write up* by Yuval and Trian https://www.kaggle.com/c/trackml-particle-identification/discussion/63313 and https://www.kaggle.com/c/trackml-particle-identification/discussion/63313#370504

[4] *Calculate angle of incidence based on cells.csv* by Jakub Gukowski https://www.kaggle.com/jakubguzowski/calculate-angle-of-incidence-based-on-cells-csv

Appendix A.

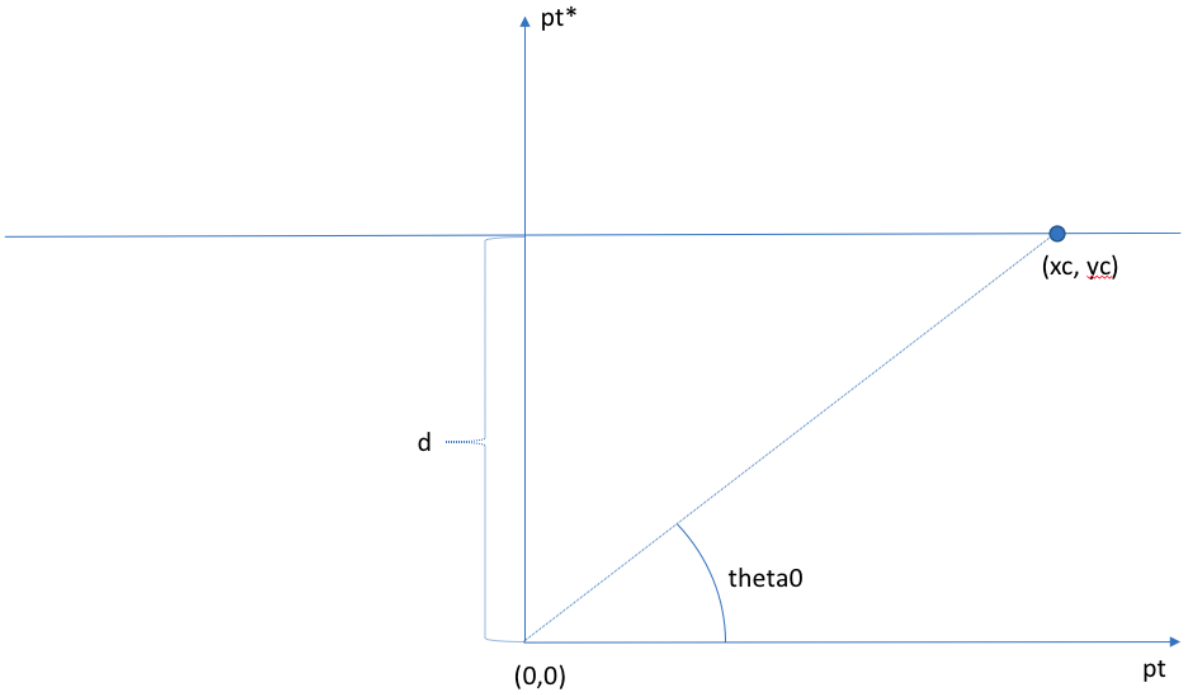
Let's use coordinates using the coordinate system defined by vector *pt*, and *pt* = (-py, px)* , its rotation by *pi/2* (see picture below). In a conformal representation, hits are represented by coordinates *(xc, yc)* where:

$xc = x / rt^2$

$yc = y / rt^2$

where

$rt = sin(theta0) / alpha0$



The distance *d* to origin of the line that goes through the point *(xc, yc)* and that is parallel to *pt* is given by:

$d = yc = y / rt^2 = sin(theta0) / rt = alpha0$

Now that we have established that relation, we can compute this distance as the projection of *(xc, yc)* on the line going through *pt*, in the original coordinate system*:

$alpha0 = (pt^* / ||pt^*||) . (xc, yc)$

Using

$(xc, yc) = (x,y)/rt^2$

we get

$alpha0 = pt^* . (x, y) / (||pt^*|| * rt^2) = (-py * x + px * y) / (||pt^*|| * rt^2)$