Fast and scalable tracking by outlier density estimation

Johan Sokrates Wind

August 2018

1 Author

Author: Johan Sokrates Wind[1]
[1] Now writing his master's thesis in deep reinforcement learning at
NTNU: Norwegian University of Science and Technology
Email: *top-quarks@protonmail.com*

2 Team in the competition

Competition Name: TrackML
Team Name: Top Quarks
Private Leaderboard Score: 0.92182
Private Leaderboard Place: 1st

3 Summary

3.1 General information

The full algorithm gave the final score of about 0.92, using about 90% of the hits. It took about 4 hours to run the model for all 125 events on a 4 core I7-7770HQ CPU with 16GB memory.

	Wall clock time	Peak memory usage
Average	7m17s	2.78GB
Max	11m20s	4.07GB

Table 1: Run-time and peak memory usage of events when running 4 events in parallel on the I7-7770HQ

The model itself was written in C++11 with only the standard library and no external libraries. A logistic regression model for candidate pruning was fit using python's scikit-learn, whose parameters were later read by the C++ code.

The training time of the model was negligible. This makes the algorithm a mixed supervised and unsupervised approach. However, the main part of the code was based on analytical derivations using statistics and 3d geometry, and acceleration data-structures for efficient implementation.

4 High-level Description

Note that I have no knowledge of the field or generally used methods, and I spent little effort in trying to look at them. Therefore I don't know what my novel findings are. However, the most important features of the algorithm might be the following:

- **High quality, efficient implementation**
  What set this algorithm apart from similar algorithms of other competitors was likely the implementation. The implementation was very efficient, enabling quick parameter tuning of the full model, quick development iterations, and larger candidate set of tracks. It also combined many different ideas like: exploiting the cell's data, modelling the magnetic field, modelling outlier densities accurately, modelling helices exactly, trying many different approaches for each step of the algorithm to see what worked best.

- **Clearly separated algorithm steps**
  Separating the algorithm into clearly separate steps enables: benchmarking each step separately both in run-time and accuracy, running single steps independently to improve then one at a time, make each step a manageable complexity, compare different approaches for each step.

- **Clear mathematical meaning of tuning parameters**
  In earlier version of the algorithm, there were many difficult tuning parameters corresponding to thresholds. This was largely resolved by a clear mathematical model of outlier densities, which had both a clear meaning, and were quick to compute in the code.

The algorithm can be very crudely summarized as follows:

1. Select promising pairs of hits
2. Extend the pairs to triples
3. Extend triples to tracks
4. Add duplicate hits to tracks
5. Assign hits to tracks

5 Scientific details

I divided my algorithm into several steps, and created a scoring metric after each step, so that I could easily tell at which step I could earn the most score. I also made load / score function after each step for rapid debugging and tuning.

There were 48 layers in the detector, each either an annulus or cylinder (approximately). I sorted these approximately so that each track would pass the layers in increasing order. I considered multiple hits of one particle on a single detector to be duplicate measurements, and only looked for a single hit per detector per track until step 4 (Add duplicate hits).

5.1 Select promising pairs of hits

This was done by considering all pairs of hits on 50 pairs of adjacent layers that covered most of the tracks. These candidates were pruned heavily by a logistic regression model of several heuristics. Some of the heuristics were how far the line passing through the two hits passes from the origin, and the angle between the direction between hits and the direction given by the cells data for each of the hits. This gave about 7 million candidate pairs covering about 99% of the score (meaning for tracks worth 0.99 had at least one pair on that track).

5.2 Extend the pairs to triples

This was done by extending the line passing through a pair, and looking where it hits the next adjacent detector layers using 3d geometry. I set the 10 closest hits to the intersection as triple candidates. Then I did another pass of pruning by logistic regression to get about 12 million candidate triples. In this step we had three points, so we could fit a helix through them, and we even had one degree of freedom left as a feature for the logistic regression. Other features were (the logarithm of) the radius of the helix, and again the deviation from the direction given by the cell data. The triples covered about 97% of the score (meaning for tracks worth 0.99 I had at least one triple on that track). And the remaining tracks were short, crooked (low momentum), and started far from the z axis.

5.3 Extend triples to tracks

We fitted a helix through the three hits, and extended it to the adjacent layers using 3d geometry. I always used the helix fitted by the 3 nearest hits on the track to the layer in question. Also here I added the closest hit to the intersection. The resulting (still about 12 million) tracks now contained about 60 million hits, and about 95% of the score (meaning if we optimally assigned tracks using the ground truth data, added all duplicate hits to each track, and ignored ¿50% coverage constraints, we could get score 0.95).

5.4 Add duplicate hits to tracks

For each track we added the hits closest to it on each layer it passed through. I'm not exactly sure how, but now we covered about 96% of the score

5.5 Assign hits to tracks

Until now all tracks had been processed completely separately, so they were massively overlapping. The goal here was to pick the best paths, and resolve any conflicts between them. My algorithm for this step was based on taking the "best" track (I will come back to the metric), removing all hits contained in it from all conflicting paths, and then repeating until there was nothing more to do. This was done efficiently using a data-structure based on a priority queue and dynamic updating of track scores.

The scoring metric to determine the "best" tracks was originally based on a random forest and distance from helixes, but I later found something much better. I didn't manage to model the perturbed helix noise. At least, I didn't feel like I had enough quantitative information to do this properly. This meant modeling the probabilities accurately as needed f.ex in a Kalman filter was infeasible. So instead of modeling the inliers (actual helix track), I modeled the probability of outliers (that we would find this track by chance). This was based on the assumption that we could model outliers by the density of hits on a layer, which I assumed was independent of the angle around the z-axis. This outlier density idea was also used for thresholding in all previous steps, so f.ex. saying "I want 0.1 outlier duplicates on average from each hit" for making the thresholding distance for duplicates.

5.6 Implementation details

Of course there were several very important implementation details, note that the above explanation is a simplification down to the most important parts. A crucial technique considering performance, was that I used an acceleration data-structure to quickly access points to close to the helix intersection with a layer. This data-structure based on quad-trees was highly efficient, supported elliptic queries, and took into consideration imprefectness of the layers (they are not exactly annuluses and cylinders), and used polar coordinates to make the maths tractable. I also made a O(1) lookup for close to analytic outlier probability densities in any elliptic region on a detector. A crude model of the magnetic field strength as function of z position of the detector ("1.002-z'3e-2-z'$\hat{2}$(0.55-0.3*(1-z'$\hat{2}$))", where z' = z/2750) gave a 0.003 score boost. On top of that there were a lot of parameters to tune, which were what gave me the last 0.01, and I'm sure there is more to gain if I had the patience.

6 Interesting findings / failed approaches

6.1 Events are qualitatively similar

I never downloaded more than then 100 first training events, and mostly only used the event1000 for testing and model fitting (the exception was some parameter tuning during the last days). Occasionally I would run the method on all the 100 events, but there were never any sign of over-fitting. This leads me to believe the events are very similar qualitatively.

6.2 The tracks starting far from the origin are the ones we need to worry about

In the beginning I used the assumption that every particle should start in the center for extending pairs to triples. Later I added another pair extension option, which was based on assuming the helices were fairly straight. To my surprise, ensambling the two approaches gave worse results than just using the straight helix assumption. This could mean that the tracks starting from the origin are very "easy" compared to the remaining tracks. Meaning if we have an approach that can capture most far from origin helices, it can probably also capture practically all the tracks starting from the origin.

6.3 It is hard to recursively remove tracks when they are found

A natural extension to the algorithm is the following:

1. Select promising pairs of hits
2. Extend the pairs to triples
3. Extend triples to tracks
4. Add duplicate hits to tracks
5. Assign hits to *only* the most probably tracks
6. Repeat from step 1. until all hits are assigned

This could simplify the search for pairs, as there are less outliers to consider. However, in practice it seems even the most probable tracks contain some wrong hits. This causes the next iteration to have problems with finding tracks missing some hits. This is a problem since both the triple and path extensions work much better for short distance extensions.

7 Faster model

It is possible to reduce run-time significantly by considering fewer candidates / do heavier pruning. Run-time and memory usage scales roughly linearly with number of considered candidates. As a very naive example, we can set "n = 5" in function findPairsNew() in new_pairs.hpp and "target = 10" in addDuplicateTriples() in main.cpp. This gives score about 0.89 in about 1 minute per event (divided by number cores when running in parallel, so 15s per event throughput) and 1GB peak memory usage.

8 Model Training and Execution Time

This is covered in the section "General information".

9 Outlook

With more time there are a few improvements that could significantly improve the algorithm. The most important one is probably the following: I spent a significant amount of time developing a DAG (directed acyclic graph) algorithm for extending and selecting tracks. This is theoretically faster and better than naively storing all paths (which may overlap significantly) and greedily picking the best one. I got to the point where the DAG method could improve the score significantly (+0.01 score compared to greedy) on simplified tests, with similar run-time. However, I didn't have time to implement all the details needed for the general case.

10 References

The implementation is available open source at https://github.com/top-quarks/top-quarks