Dynamic parameter optimization of a HMM used in a lap counting algorithm

Abstract

In this report I will discuss the optimizations that can be made to the current lap counting system used during the 12urenloop event. This system makes use of a manually configured HMM. I will discuss some limitations of the current algorithm and then discuss how I plan to overcome these by training the HMM using Baum Welch. Finally, I am able to improve on the results achieved during last year's event.

1. Introduction

Before I start with the technical aspect of my project, I will quickly introduce the setting for my project.

1.1. 12urenloop

12urenloop (12urenloop, 2022) is a yearly event where teams try to run as many laps as possible in a timespan of 12 hours. Each team can only have one runner at a time. The event takes place in the center of Ghent at a track of about 300 meters.

An essential part of the event is tracking the lap count of each team, in order to be able to crown a winner. For a long time my student union Zeus WPI (Zeus WPI, 2022) has taken care of this challenge. We strive to provide an automatic system that once setup can count the laps for 12 hours, so we don't have to do this manually. We also provide a site where the live score can be followed during the event.

Since 2021 we also provide lap counts during the 6urenloop (6urenloop, 2022) event in the center of Bruges. The main difference for our system between the 12urenloop and the 6urenloop is the track length. The 6urenloop has a track with a length of about 150-200 meters.

Last year, we reworked our whole setup from the ground up, in an attempt to make our system easier to work on and more extendible for extra features.

1.2. Setup

In this section, I will explain how our current setup works, from detection where runners are to a live count on our live site. A schematic overview is shown in figure 1. All components will be discussed in more detail later.

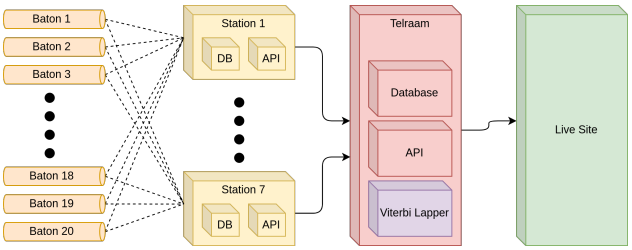

Figure 1. A schematic overview of our setup

1.2.1. Batons

Batons (orange in figure 1) are essentially special relay batons used by the teams during the race. Each baton is equipped with a Bluetooth transmitter that broadcasts a message about 8 times per second. This message contains key information like the baton identifier and some monitoring information like baton uptime and battery level.

1.2.2. Stations

Stations (yellow in figure 1) gather the messages sent by the batons. They save all information transmitted by the baton with additional data like the timestamp and the Received Signal Strength Indicator (RSSI). I will refer to such a message as a detection for the rest of the report. The data is saved in a local database that is then in turn exposed by an API. This allows the next component to gather the data of all stations.

In contrary to the batons that circle the track with the runners, the stations are positioned around the track at fixed places. They are placed as equally distanced from each other as possible.

1.2.3. Telraam

Telraam (Telraam, 2022) (red in figure 1) is the central component of our system. It is an application written in Java that does all the actual processing of the detections. This component performs 3 main tasks.

- Gathering all data from the stations by querying all their APIs and saving the result in the local central

database. Telraam only adds one more piece of information to the detections, namely the station it was detected at. Detections now look like this in the database:


```
{
    'id':  3493712,
    'batonId':  4,
    'stationId':  2,
    'rssi':  -95,
    'battery':  95.0,
    'uptimeMs':  31684964,
    'remoteId':  267237,
    'timestamp':  1651071359000
}
```


- Housing the lap counting algorithm, the algorithm that takes the detections as input and returns the laps a team has run. We have 1 main algorithm (also referred to as lapper) which uses Viterbi. This algorithm is explained later in the report.
- Provide an API to expose the data and control vital configuration of the system. For example, which baton is assigned to which team.

1.2.4. Live site

This uses the lap count exposed by the Telraam API to provide a live overview of the current score. A screenshot of this page is shown in figure 2.

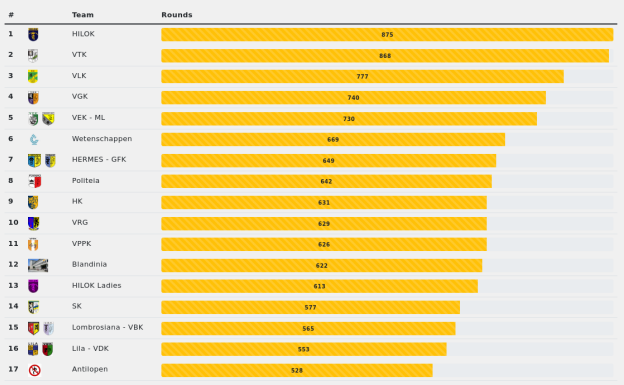

Figure 2. The final scoreboard during the 12urenloop 2021

1.3. Viterbi Lapper

Here I will discuss the main algorithm referenced before. This will also be the starting point of my project.

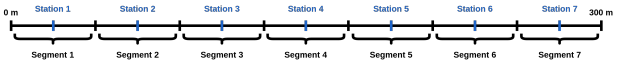

Figure 3. A schematic overview of the segments on the track

The current algorithm uses the Viterbi algorithm. The observations at certain stations act as the emissions. The hidden states are segments of the track around the stations. For example, we let segment 1 represent the 20 meters before and after station 1. This is visualized in figure 3 for all 7 segments. We try to capture this idea in the transition and emission matrix used for generating the Viterbi path. When creating the matrix with transition probabilities, we can generalize this to the following structure. For a segment n the runner is most likely to stay in the same segment, a slight chance to move to segment $n-1$ or $n+1$ and even less chance to move to segments $n-2$ or $n+2$. The same can be said for $n-3$ and $n+3$. With this in mind, we designed a matrix with roughly the shape as can be seen in figure 4.

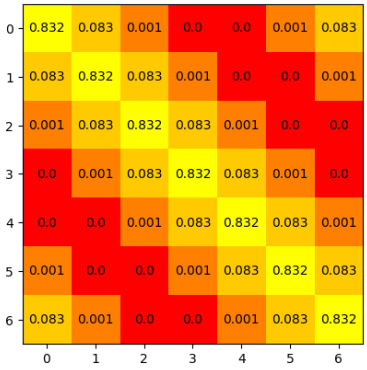

Figure 4. Visualization of the shape of the transition matrix used in the HMM

The same line of reasoning can be followed when designing the emission matrix. We will most likely receive detections at station n when we are at segment n. With this in mind, we designed the emission matrix to follow the structure as visualized in figure 5.

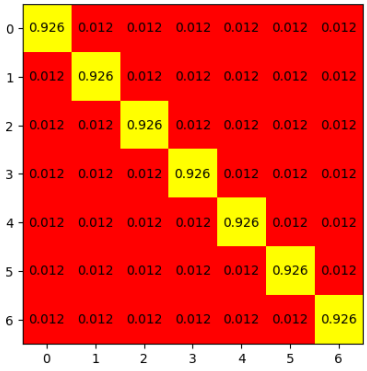

Figure 5. Visualization of the shape of the emission matrix used in the HMM

While the structure of these matrices will be fixed, the ex-

act values will not. During the 6urenloop for example, the track will be much shorter and there will be more overlap at which station a runner is detected. Because of this, we have to manually tweak these parameters during the first hour(s) of the event in order to find parameters that work for the current setup of the stations around the track.

Using these matrices we then find the Viterbi path given the detections. Using the found path we then deduct the amount of laps using a simple algorithm that essentially just count how many transitions from segment 7 to 1 we have. This algorithm is simple but robust against several edge cases that we could encounter in the Viterbi path.

In Telraam we only have detections with second precision. So some filtering is applied on the detections. For all detections of a single team that happened within the same second, we only keep the one with the highest RSSI. Additionally, we only look at detections with a RSSI higher than -75. This reduces the noise in the data a lot.

Additionally we also monitor the laps generated by the algorithm. If we notice there is a missing lap we manually add an extra lap to the team.

1.3.1. Improvements

The fact we have to configure these parameters manually at the start of the event is of course not ideal. Because during this time we will be counting manually as a backup. Only when we find the right parameters we can stop counting manually. It would be very nice to be able to automate and speedup this process by fitting the parameters automatically.

By finding these parameters automatically we can hopefully improve on the following things:

- The parameters will be optimized for the current track layout. These parameters being less generic will improve our accuracy of the Viterbi path.
- When a station goes down we can quickly react by fitting the parameters with the observations minus the detection of the broken station. The parameters will then again be optimized for the new situation.
- The amount of laps we have to manually correct. In essence, this comes down to improving the accuracy of the path generated by Viterbi using our HMM.

In order to fit the detections to the HMM I would look into using the Baum Welch algorithm (Baum–Welch algorithm, 2022). Specifically because this works on unlabelled data, which is ideal for our use case.

2. Java Baum Welch

2.1. Idea

My initial idea was to try to implement Baum Welch myself in Java for 2 main reasons.

- Telraam is written in Java, so also using java for Baum Welch would make integration a lot easier.
- No good existing implementations could be found.

2.2. Troubles

Implementing this correctly is hard. A lot of mistakes can be made that are hard to pin down when debugging. This is why I could not get my implementation to work easily. After quite some time making slow progress, we decided during the intermediate presentation it would be more efficient to look towards other existing implementations in other languages, at the cost of easy integration into the existing codebase.

During the implementation however, I encountered an issue I would still like to discuss.

The Baum Welch algorithm consists of 3 main steps.

- **The forward procedure**: computes the probability of ending in a certain state given the first n observations of the sequence. (Forward-backward algorithm, 2022)
- **The backward procedure**: computes the probability of observing the remaining observations given a starting point n (Forward-backward algorithm, 2022)
- **The update step**: In this step we update the transition, emission and start probabilities. (Baum–Welch algorithm, 2022)

However, if n is large enough (100 is more than enough) we encounter a problem. The probabilities in the forwards series converges exponentially to zero. When the probability becomes 0 or NaN we lose all meaningful information. In my problem I am working with close to 2 million observations, so this will definitely cause issues. Because of this, I had to implement some scaling to prevent this. Specific details on how to do this were very hard to find. I tried to implement scaling using an online post (cwl, 2022) I found. However, after I implemented this, my implementation was not converging to a solution. So in the end, I had no working implementation and as discussed, started looking for existing alternatives.

3. Python HMMlearn

As discussed in the last section, I moved to an existing implementation of the Baum-Welch algorithm. I chose the

Python hmmlearn library (hmmlearn, 2022). It is a specialized and tested library with about 2700 stars on GitHub. It builds on Scikit-learn and NumPy. The library was easy to use and fast. The only downside I noticed was the lack of complete documentation. Only the more complex HMM models were documented online. I had to find the one I needed in the source code.

3.1. Fitting the data

To fit my observations to the HMM I used the CategoricalHMM class provided by hmmlearn. As initial transition and emission matrix I used those described earlier, in an attempt to optimize those. The training data used in the Baum Welch training process is the same filtered observations as described in section 1.3. The observations itself came from the 12urenloop event in 2021.

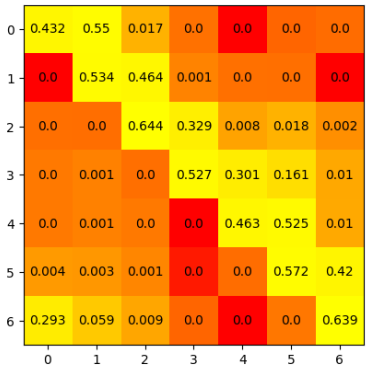

Figure 6. The optimized transition matrix used in the HMM

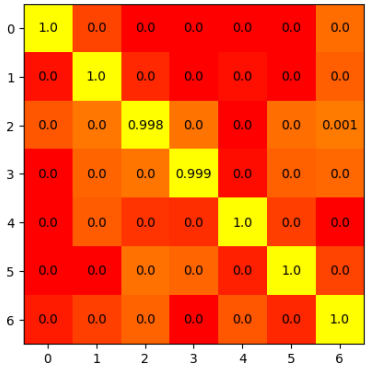

Figure 7. The optimized emission matrix used in the HMM

This time the training converged in about 35 iterations of the Baum Welch algorithm. This took a few seconds to optimize. The optimized matrices for this data can be seen in figure 6 (the transition matrix) and figure 7 (the emission matrix).

There clearly is a difference between the initial matrix in figure 4 and the updated matrix in figure 6. This can mainly be contributed to the Baum Welch algorithm that, given our detections, learned that runners only run in one direction around the track. The form of the emission matrix stayed more or less the same. It only lowered the chance to be detected at a station n when in segment m, with $n \neq m$, by a lot.

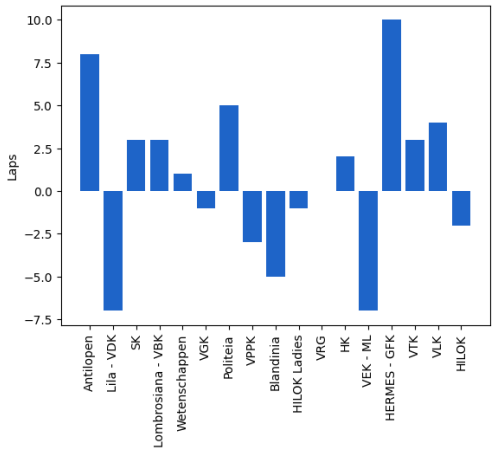

Figure 8. Difference of results between original parameters and new parameters of the Viterbi algorithm. Positive numbers mean the new parameters generated more laps than the original. Negative the other way around.

When looking at the final lap counts I do not notice a big difference. The results are about the same with no decisive difference in one direction. The new parameters awarded slightly more laps. The difference is visualized in figure 8.

However if I look at the results obtained compared to the final scoreboard during the 12urenloop 2021 (so original Viterbi output + manual adjustments), there are still missing a lot of laps that even our new parameters could not pick up. This is visualized in figure 9. The attempt to further improve this are discussed in the next section.

3.2. Improving the data

In an attempt to further improve the accuracy of the HMM in order to limit the amount of extra manual adjustments that have to be made to the generated laps, I looked towards 2 factors that I believed had the biggest impact on the model's performance:

- The fact we only have detections in Telraam with second precision, causing us to filter away a lot of detections. The batons send messages 8 times per second, which means we could benefit of having more precision.
- Maybe the -75 RSSI filter is too strict, so I lowered

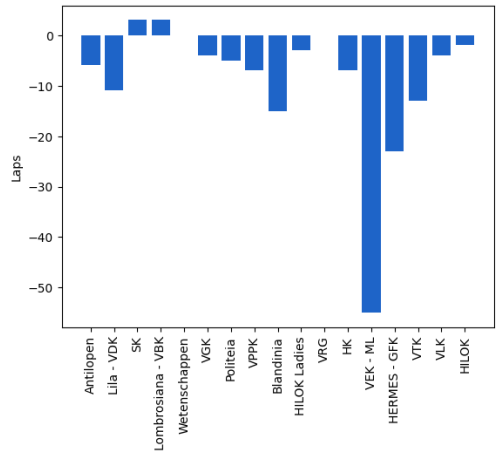

Figure 9. Difference of results between the new parameters of the Viterbi algorithm and the final scoreboard of the event in 2021. Positive numbers mean the new parameters generated more laps than the final scoreboard. Negative the other way around.

this to -80.

When looking into the first point described above, I noticed we only have second precision in Telraam. In the stations the timestamp of the message was saved as a float with millisecond precision. So I obviously had some loss in precision when transferring the data from the stations to Telraam. It turned out the issue was a cast from a `Float` to a `Long` which caused the timestamp to be rounded, losing the millisecond precision in the process. I fixed this and then reimported the data.

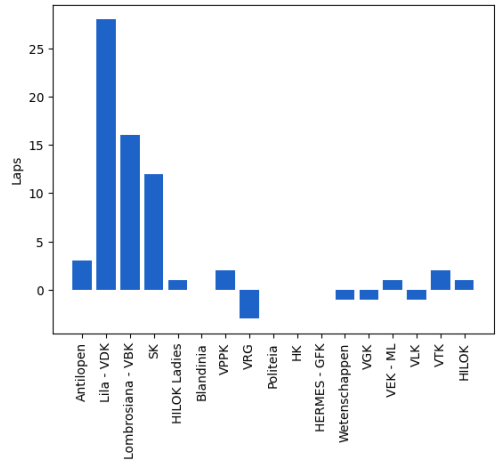

Figure 10. Difference of results between the new parameters of the Viterbi algorithm and the final scoreboard of the event in 2021. Positive numbers mean the new parameters generated more laps than the final scoreboard. Negative the other way around.

With these changes the final result is much better. Training took a few seconds more because fewer observations were filtered out. The difference between the generated laps and the final scoreboard during the 12urenloop is much less for the top teams. The difference is at most 2 laps. This is a huge improvement compared to the results before these changes. Now only minimal manual corrections will be required. This is again visialized in figure 10.

In figure 10 there is a big noticeable difference with the teams at the lower end of the leaderboard. This can probably be attributed to the fact that during the past 12urenloop event, we mostly monitored the teams higher in the leaderboard and missed some laps for the teams at the bottom of the leaderboard. This can be visually verified by plotting the lap times using both the old and new (trained) parameters.

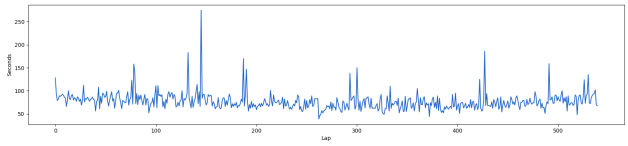

Figure 11. Seconds between consecutive laps for the ”LILA - VDK” team, using the old parameters on the filtered observations before the improvements were added.

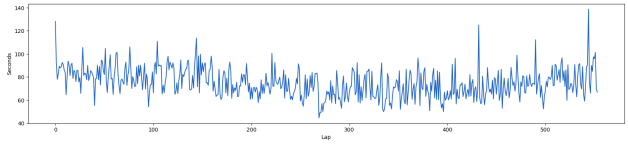

Figure 12. Seconds between consecutive laps for the ”LILA - VDK” team, using the trained parameters on the observations with improved filtering.

In figure 11 we see some distinct peaks in lap times. These are the laps we missed adding manually. These outliers are gone or drastically decreased in figure 12.

Next to these great results, I again ran into the importance of this project. The manually configured parameters are not working any more for the new filtered observations. lap counts using these parameters are now more than 100 laps off. While I can now simply train my model from these parameters in a few seconds to achieve the best results yet.

4. Further steps

Next few months I will look into integrating my Python implementation for this paper with Telraam. So we can use the improved algorithm during the next event.

Some other ideas I had but lacked the time to research are the following 2:

- Using 7 segments in the algorithm make it easy to reason about the shape of the transition and emission matrices, but are probably not required for accurate lap counts. Using 3 segments should be enough to count laps. Reducing the segments to 3 will probably also have a positive impact on training speed and the speed in which the Viterbi path can be constructed. We however want to keep using the 7 stations as emission states. This gives us some redundancy in the amount of stations that can break, but the amount of hidden states does not impact this.

- In the other direction, it would be interesting to test how feasible the idea of using more than 7 segments (for example 14 or 28) is. The goal would not be to use this for the lap counting algorithm, but rather for a visualization. If we divide the track into more segments, we can more accurately predict the position of a runner on the track, which might be interesting to use in a visualization.

References

12urenloop. 12urenloop, 2022. URL https://12urenloop.be. [Online; accessed December 2022].

6urenloop. 6urenloop, 2022. URL https://6urenloop.be. [Online; accessed December 2022].

Baum–Welch algorithm. Baum–welch algorithm — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm. [Online; accessed December 2022].

cwl. Scaling step in Baum-Welch algorithm, 2022. URL https://stats.stackexchange.com/questions/274175/scaling-step-in-baum-welch-algorithm. [Online; accessed December 2022].

Forward-backward algorithm. Forward-backward algorithm — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/wiki/Forward%E2%80%93backward_algorithm. [Online; accessed December 2022].

hmmlearn. hmmlearn — Hidden Markov Models in Python, with Scikit-learn like api, 2022. URL https://github.com/hmmlearn/hmmlearn. [Online; accessed December 2022].

Telraam. Telraam — New and hopefully improved application to count laps of the 12urenloop event, 2022. URL https://github.com/12urenloop/Telraam. [Online; accessed December 2022].

Zeus WPI. Zeus WPI — The student association for Computer Science at Ghent University, 2022. URL https://zeus.gent. [Online; accessed December 2022].