Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,
Arbeitsgruppe Collective Intelligence and Biorobotics

Feature Engineering and Probabilistic Tracking on Honey Bee Trajectories

Franziska Boenisch

Matrikelnummer: 4821885
franziska.boenisch@fu-berlin.de
Betreuer & 1. Gutachter: Prof. Dr. Tim Landgraf
2. Gutachter: Prof. Dr. Raúl Rojas
Berlin, 21. Februar 2017

Abstract

Honeybees are a popular model for analyzing collective behavior. The BeesBook system was developed to track the behavior of all individuals in the colony over their entire life spans. The bees are marked with a unique bar code-like marker, the comb is recorded, and the images are evaluated automatically to detect, localize, and decode the markers. This thesis focuses on the development of a tracking method, i.e., linking detections through time, and a method to determine correct ids for the resulting tracks. As a result, a random forest classifier was found to perform best to decided if two detections should be linked. The features implemented for it are based on id similarity of the detections and a simple motion model. Evaluation results show that the tracking merges about 90 % of the tracks correctly. The id assignment returns correct ids for more than 85 % of all computed tracks or track fragments.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 21.02.2017

_____

Franziska Boenisch

Acknowledgements

I want to thank everyone who has accompanied me during the time I wrote my thesis and supported me directly or indirectly.

First of all, I take this opportunity to express gratitude to the people involved in the BioroboticsLab. Their creative ideas and help for each other create a very productive and homelike environment. I especially want to thank my supervisor Dr. Tim Landgraf for his excellent support and his supervision. The door to his office was always open whenever I needed some advice about my research and writing.

Furthermore, I want to thank Jonas Cleve for his encouragement and support during the last months that helped me through some struggles with the thesis. I also want to thank him for his thorough review and his comments that significantly improved my manuscript.

I am also profoundly grateful to my parents Sabine Wiebe and Peter Boenisch for providing me with support and encouragement throughout my years of study and through the process of researching and writing this thesis.

Finally, I want to thank the German National Merit Foundation (Studienstiftung des Deutschen Volkes) for their financial support. Without their scholarship, I would not have been able to work as dedicated on this thesis.

Contents

1 Introduction **1**

2 Related Work **5**
- 2.1 BeesBook Setup . . . 5
- 2.2 Software Pipeline . . . 6
- 2.3 Tracking Framework . . . 9

3 Implementation and Evaluation **13**
- 3.1 Tracking Improvement . . . 13
  - 3.1.1 Training Data Generation . . . 13
  - 3.1.2 Classifier Selection . . . 17
  - 3.1.3 Feature Extraction . . . 22
  - 3.1.4 Feature Selection . . . 27
  - 3.1.5 Hyperparameter Tuning . . . 33
- 3.2 Id Assignment . . . 36
  - 3.2.1 Euclidean Distance Weight . . . 41
  - 3.2.2 Rotation Distance Weight . . . 42
  - 3.2.3 Position Weight . . . 44
  - 3.2.4 Confidence Weight . . . 45

4 Discussion **47**

1 Introduction

A honeybee colony is an interesting example of a complex and dynamical system [38]. The interaction and communication of thousands of individuals enable the hive to adapt to changing conditions in a parallel manner. These conditions can, for example, include variations in temperature, the arrival of intruders to the hive, and food shortage. One form of communication between the bees is the so-called waggle dance. It is performed to inform each other about food sources and nest site locations. It is already well investigated how the properties of the food source are reflected in the movements of the dancer bees [38]. However, the effect on the followers, the integration of the dance into the whole communication process, as well as other communication mechanisms remain unexplored.

The BeesBook project described by Wario et al. [39] developed a setup consisting of hardware and software to track all bees in a hive automatically. It is the first system which makes it possible to investigate the communication further and to explore how previously gained experience of interaction partners influences their interactions. The automatic approach removes human bias from the analysis.

The first part of the system consists of marking the bees with individual tags. The observation of the hive is then done by cameras that take pictures of the comb. The images are evaluated automatically to detect and locate the tags and decode their ids. As the tags are not always perfectly readable by the software, their ids might be decoded wrongly. Therefore, the detection ids cannot be used to link detections to tracks, representing the movement of each individual over time. Instead, a probabilistic tracking performs the task of assembling the tracks. It is organized hierarchically. A first iteration merges detections into track fragments. All further iterations connect those fragments to complete tracks (see Figure 1.1). In the end, the correct ids of those tracks are determined.

So far, the first iteration of the tracking works well. However, when it comes to merging the resulting fragments, the tracking still produces many errors: either it fails to connect fragments that belong together, or it wrongly merges fragments from different tracks. In the first case, the resulting fragments are short and do not provide sufficient information for the analyses. In the latter case, they mix information from two different bees and might thereby lead to wrong analysis results. To avoid this, a refinement of the tracking based on fragments is necessary. This thesis provides such an improvement by selecting

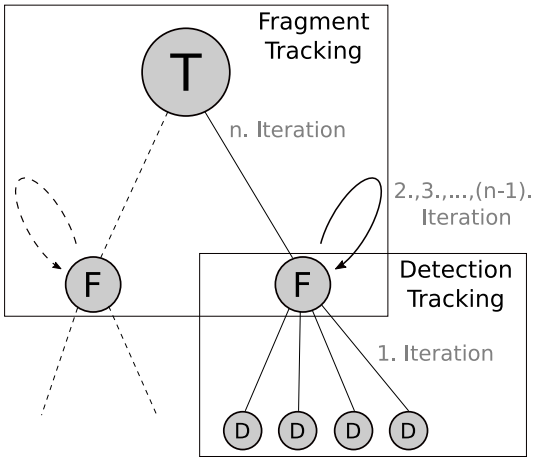


Figure 1.1: A schematic overview of the hierarchical organization of the tracking. In the first iteration, the detections (D) are merged into track fragments (F). In all further iterations, those fragments are iteratively connected until they represent a complete track (T) of an individual.

a supervised binary classifier, that is trained to decide if two given fragments should be combined or not. Therefor, labeled training data was generated, and complex behavior features were implemented as an input to the classifier. In the end, a well-performing feature subset and the hyperparameters that produce the best classification results were determined. Due to the wrongly decoded tag ids, even correctly merged tracks can contain several different ids. This thesis also presents a solution to the determination of the correct track id.

These improvements lead to longer and less erroneous tracks with correctly assigned ids. Such tracks make the following analyses of the communication in the hive possible:

- An entire lifetime of a single bee can be analyzed. This includes her motion, her tasks that might change over time, and her interactions with other bees.
- The correctly determined ids can be used to analyze not only individuals but also social networks within the hive.
- It becomes possible to investigate dancers' and followers' specific preferences concerning the waggle dance over time. These preferences can include, for example, particular locations on the comb, dance properties or even particular bees [39].

The next section provides an overview of the related work with a particular focus on the already implemented parts of the BeesBook project. Section 3.1 successively describes

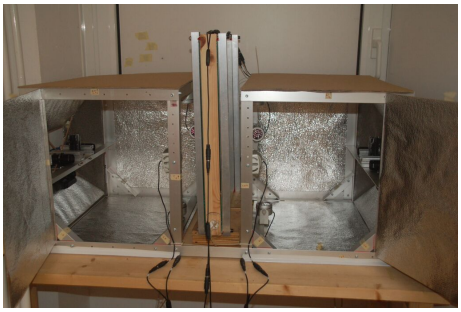
the generation of training data, the selection of a classifier, the extraction of features, the selection of a minimal well-performing feature set, and the hyperparameter tuning of the selected classifier. In Section 3.2 several approaches to an advanced id assignment are introduced. The discussion of the improvements and an outlook for further work are provided in Chapter 4.

2 Related Work

An experiment, very similar to the BeesBook project, was conducted by Mersch, Crespi, and Keller [29] in 2013. They performed tracking on an ant colony to investigate the insects' behavior. Therefor, they developed an observation setup and installed unique quadratic tags on the ants' backs to distinguish the individuals. The tags encode 36 bits of information, thereof 26 for error correction [17]. This tag design leads to a very high accuracy in the id determination such that no probabilistic tracking is needed. However, neither the setup nor the tags can be utilized in the BeesBook project. The setup is not suited to hold the two-sided honeycomb built by the bees to contain their larvae and to store honey and pollen. The quadratic tags would hinder the bees in their movements. Hence, a setup and a probabilistic tracking needed to be developed specially for the BeesBook project. They are described in the following three sections.

2.1 BeesBook Setup

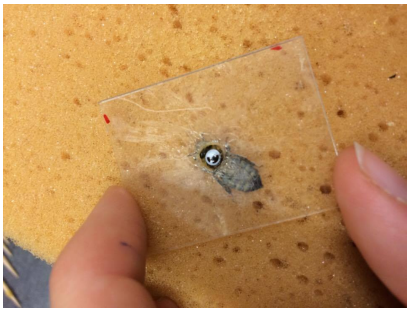
The BeesBook setup uses a modified one-frame honeybee observation hive (see Figure 2.1b) standing inside an aluminum scaffold (see Figure 2.1a) that holds infrared LED cluster lamps and six cameras [39]. Two of the cameras are used for a real-time



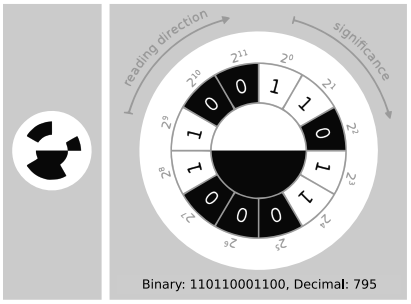
(a) The hardware setup consisting of a comb, located inside an aluminum scaffold that is equipped with cameras and infrared LED lamps.

(b) The observation hive populated by tagged bees is covered by a glass panel that keeps the bees from flying around inside the recording setup.

Figure 2.1: The recording setup.



(a) A bee during the attachment of the circular tag attached to her thorax.



(b) The design and interpretation of the circular tags.

Figure 2.2: The tags.

waggle dance detection. The other four cover the two sides of the hive, two cameras per side. Each camera takes three frames per second. Infrared light, which is not perceivable by bees, is used for illumination.

To identify the individual bees, unique tags need to be attached to their thorax (see Figure 2.2a). The tags are circular and curved. This particular design exploits the maximal space on the bee's thorax and does not hinder her from activities like flying and entering comb cells. The tags consist, as depicted in Figure 2.2b, of an inner and an outer circle. The inner circle is divided into two halves to indicate the bee's orientation. In the tracking setup, the white half is orientated towards the bee's head, showing the front direction. The outer circle is divided into 12 segments representing one bit of information each. The 12 segments are a binary code for the bee's id. With 12 bits, $2^{12} = 4096$ bees can be uniquely identified. More bits of information could be useful for error correction, but several factors are limiting the possible number of bits on a tag, like the size of the bee's thorax, the resolution of the records and the exposure time [30].

The lack of error correction causes wrongly decoded ids. It is, thus, not possible to track bees over time by their ids. Instead, a probabilistic tracking is needed. The data used by the tracking is generated from the image data by software that implements several processing steps. This software is called *pipeline* due to its organization.

2.2 Software Pipeline

Figure 2.3 shows the location of the pipeline in the BeesBook project. The images captured by the recording setup are compressed on-the-fly to videos containing 1024 frames each. The video data is then transferred to a large storage from where it can be

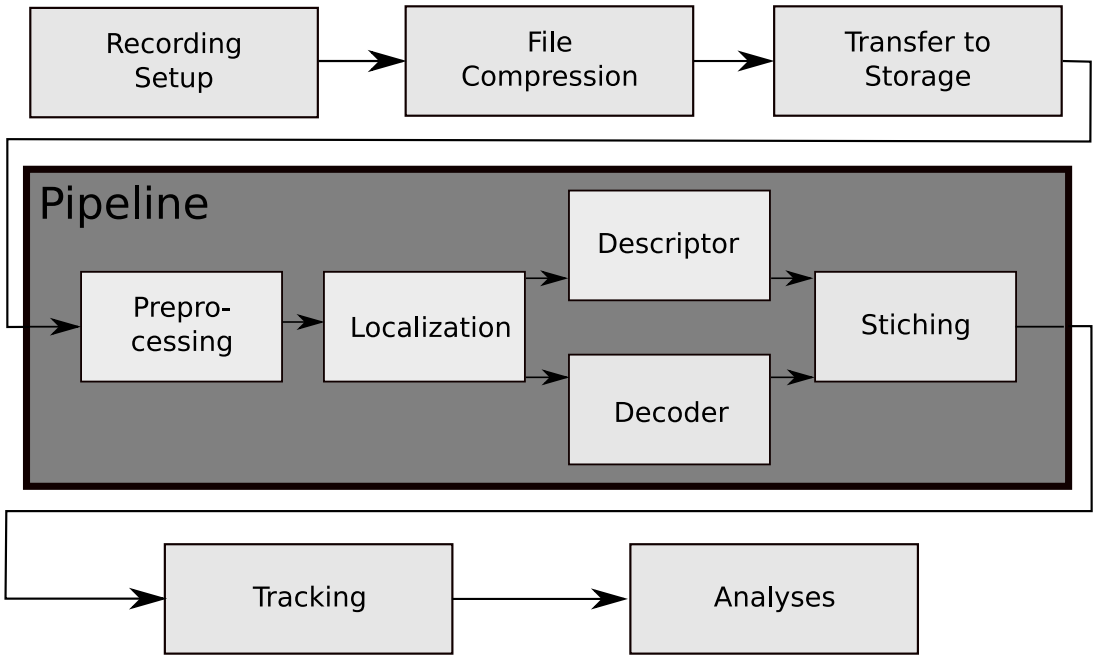


Figure 2.3: A schematic overview of the BeesBook processing pipeline.

accessed by the pipeline for processing. This processing, which is not performed in real-time, consists of the following steps:

1. *Preprocessing:* The preprocessing step prepares the images by doing a contrast adjustment. This is needed to compensate for illumination variance.

2. *Localization:* In this step, the tags are localized in the image by a neural network.

3. *Decoding:* A second neural network decodes the tag ids, rotations, and positions.

4. *Description:* A third neural network generates a descriptor for each tag. The descriptor is a 128 bit long binary mask, that is extracted from an internal representation of a deep autoencoder [28]. It provides a low-level representation of the tag which can be used to measure tag similarity.

5. *Stitching:* As two cameras that cover different areas of the comb are used for each side, the images need to be stitched together to form a complete view on the comb. Therefore, the image coordinates of the tags are transformed to hive coordinates. The stitching also needs to deal with the overlapping areas and the resulting duplicated data.

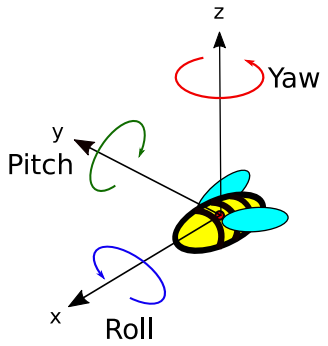


Figure 2.4: The three rotation axes of a bee [42]. The x-rotation in our setup indicates the *roll* of a bee. The y-rotation, the so-called *pitch*, appears, for example, when a bee leans into a comb cell. The z-rotation, also called *yaw* indicates the bee's orientation on the hive, i.e., the direction in which the bee looks.

The stitched data is saved in a BeesBook specific data format (`bb-binary`) whose specification is available on GitHub [5]. It relies on a Cap'n Proto [36] schema. Cap'n Proto is a very fast binary data interchange format that generates classes with accessors methods from the fields that are specified in the schema. Each tag is represented in this format as a data point called a *Detection*. A Detection contains the following information:

- A sequential index to refer to detections in an image,
- x- and y- coordinates of the detection,
- the bee's rotation on all three axes, see Figure 2.4,
- the radius of the tag, measured in pixel,
- a score to indicate how certain the localizer is in its determination of the detection's x- and y-coordinates in the hive,
- the descriptor, and
- the decoded bee id in the form of continuous scores for each of the 12 bits represented by the tag. The scores indicate how confident the network is that a certain bit is set (0 shows that it is confident a bit is not set, 255 means that it is sure that a bit is set). These numbers are later converted to percentages with $0 = 0\,\%$ and $255 = 100\,\%$.

In the `bb-binary` format, the detections are organized in so called *frames*. Each frame contains all detections from one picture. The frames with their detections are the input for the *Tracking*, which is described in the following section.

2.3 Tracking Framework

A tracking aims to connect single detections from different frames (pictures) to trajectories that allow to trace bees over time. Figure 2.5 visualizes a minimalistic example of combining detections into tracks. In the following chapters, a trajectory is referred to as a *Track* or a *Fragment*. A track is a complete trajectory that covers the whole time from a bee's first appearance on a picture until the last one, which might ideally be her entire lifetime and a fragment is only a part of a track. Sometimes, however, the notion of track and fragment are used interchangeably especially in situations of uncertainty if a certain fragment is already a complete track or if other fragments belong to the same bee.

Forming fragments by linking detections with their successors is a complex task. Under ideal conditions, there would be no gaps in the tracks, and each detection would find its successor in the next frame. With n detections in a first frame and m in a second one, $n \cdot m$ detection pairs would need to be checked to find the correct matches. However, in reality, the successor of a detection is not always located in the next frame and tracks can contain gaps. This is the case when some bees' tags are not visible in some frames, for example, when the bees work inside comb cells or are covered by other bees. It is also the case when the pipeline works inaccurately and does not detect a certain tag.

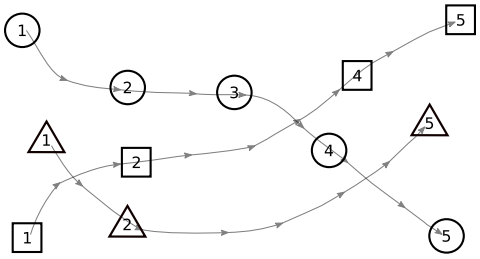


Figure 2.5: The general idea of tracking: In the picture, the detections of the same bee are represented by the same shape. Each number indicates a timestamp. The tracking aims to connect all individual detections belonging to the same bee in the right order so that they form a track. It is important to notice that the tracks can contain gaps, which means, that at a certain time, the detection belonging to the track is non-existent.

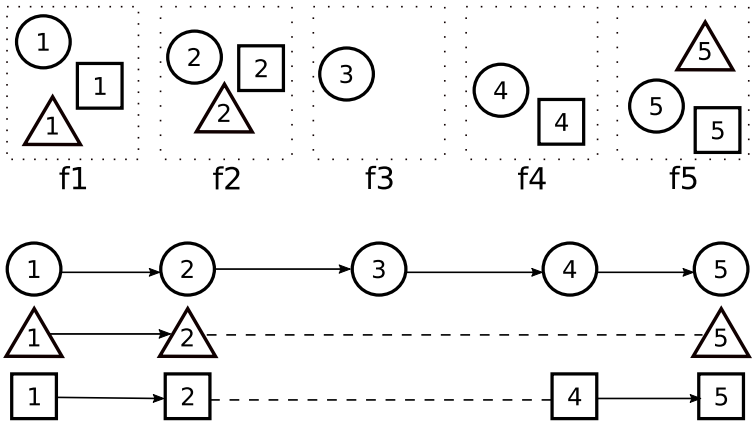


Figure 2.6: The first two iterations of the tracking is visualized. It starts with several detections organized in frames. In the picture, there are five frames, f1-f5. The first iteration of the tracking aims to connect detections from neighboring frames (in this picture indicated by arrows). The second iteration connects the resulting fragments to tracks (here indicated by dashed lines) that might contain gaps.

In these cases, the successor of a given detection might be found some frames later. To deal with this, all detection pairs from a certain number of frames would need to be checked. In the BeesBook system, an average frame contains about 200 detections. Therefore, allowing pairs of detections originating from more than two different frames is computationally too expensive.

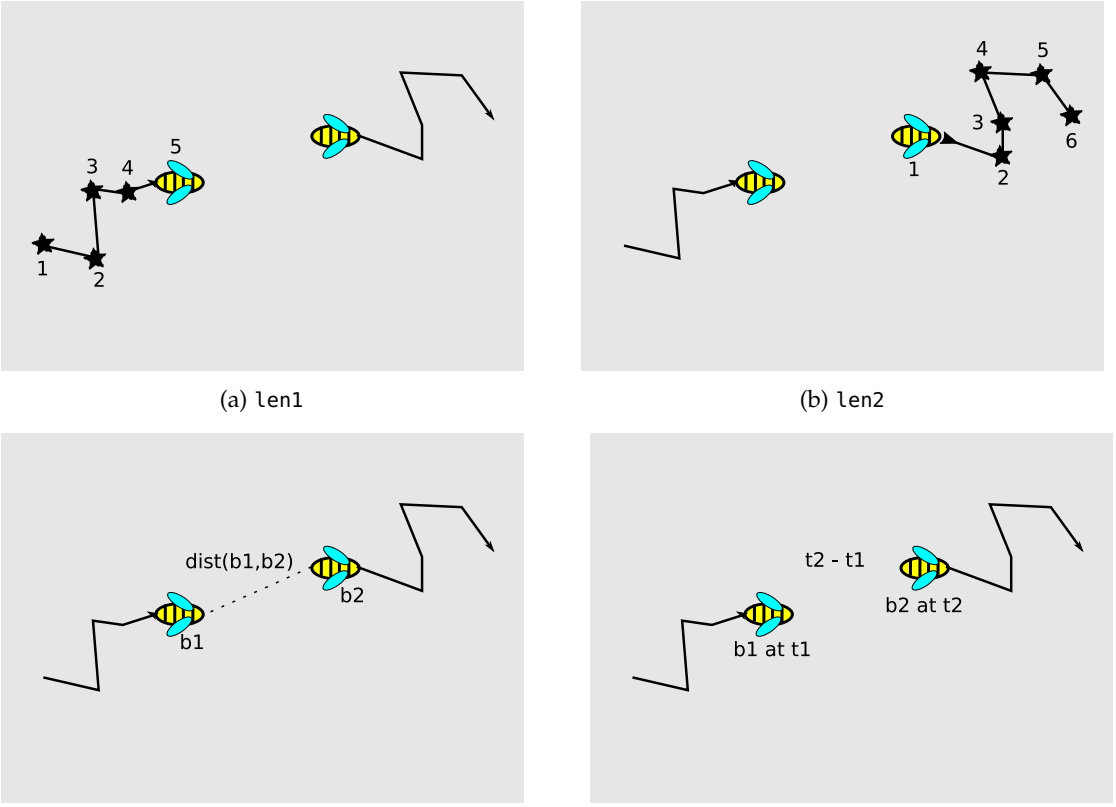
To address this issue, a hierarchically organized tracking framework has been implemented by Rosemann [35][1]. It uses an iterative approach that is visualized in Figure 2.6. In the first iteration, the tracking connects only detections from directly adjacent frames and produces fragments without gaps. Each fragment can contain several detections of which all but last one have a successor. Thus, the number of detections per frame for which a successor needs to be found is reduced dramatically. This allows the next tracking iteration to consider fragments that start more than one frame later for adherence without increasing the computational complexity too much. With each further iteration, the number of detections that need a successor is reduced again (as fragments are merged together) such that the gap size can be increased iteratively.

In the tracking framework, a supervised binary classifier takes the decision if a detection (or fragment) is appended to an existing detection (or fragment). The classifier and the features used for the first iteration of the tracking perform well. However, the features lose their predictive power with longer gaps between the fragments. Therefore,

[1]The implementation can be found in the BioroboticsLab's GitHub repository [6].

a different classifier and feature set are used for further iterations. The framework currently uses a random forest classifier [27] and the following five features to decide if two fragments should be linked.

- `len1`: Length of the first fragment, the one that another fragment can be appended to (see Figure 2.7a).
- `len2`: Length of the second fragment, which is a possible candidate to be appended (see Figure 2.7b).
- `id_distance`: Difference between the two fragments' median ids.
- `distance`: Euclidean distance between the last detection in the first fragment and the first detection in the second one (see Figure 2.7c).
- `time_diff`: Time elapsed between the end of the first fragment and the beginning of the second one (see Figure 2.7d).



(a) `len1` (b) `len2`

(c) `distance` (d) `time_diff`

Figure 2.7: Visualization of the simple features

The last step in the tracking framework is the id assignment to the computed tracks. Currently, the median id is calculated for each of the 12 bits over all detections of the track. A threshold of 0.5 is defined, and for every median value above this threshold, the specific bit is said to be set, the others are not set. The resulting binary string represents the track id.

The results obtained with this model lack refinement. Thus, this thesis improves the tracking of fragments by choosing a better classifier, doing feature engineering, and hyperparameter tuning. Furthermore, a more sophisticated approach than the median id calculation is developed to assign ids to the tracks .

3 Implementation and Evaluation

After defining the problem the tracking needs to solve and locating it in the entire BeesBook project, this chapter focuses on the improvement of its second iteration, the merging of fragments. It furthermore presents a more sophisticated id assignment to the merged tracks. Implementation and evaluation are presented together, as every step of the improvement depends on the results of the previous one.

3.1 Tracking Improvement

The tracking is done with a machine learning model. The following sections describe the model selection process and its improvement through a hyperparameter tuning. As the performance of the model depends also on the quality of the input data, the next section focuses on the training data generation.

3.1.1 Training Data Generation

The supervised binary classifier that is used to link fragments requires labeled data for training. In the second tracking iteration, the classifier should decide whether two fragments belong together or not. It, therefore, needs training data in the form of two learning fragments and a class value that shows if the second one should be appended to the first one (class 1) or not (class 0).

In the BeesBook system, there exist two hand-labeled truth datasets originating from the season of 2015. *Dataset one*, from September 18^{th}, contains detections of 402 frames. 201 frames starting a 9:36am are captured by camera 0, the other 201 frames starting at 2:51am by camera 1. *Dataset two* of September 22^{nd} contains 200 frames captured by camera 0 at 11:20am and 11:40am. The detections of both datasets are manually merged into tracks and assigned the tracks' correct ids by a person through the editor-GUI developed by Mischek [30].

This section proposes a method for generating learning fragments from those truth tracks. But before performing the data generation, the maximal gap size allowed within

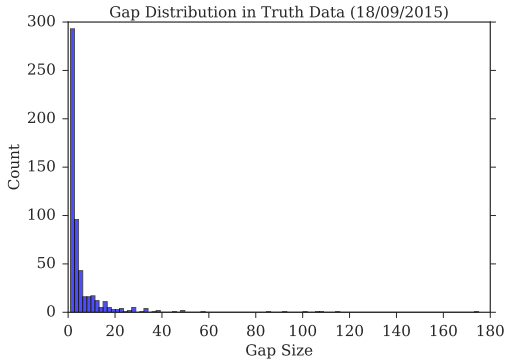


Figure 3.1: A distribution of the gap sizes in the manually determined tracks of truth dataset one.

the fragments for this tracking iteration needs to be defined. In the previous one, only detections from directly neighboring frames were linked. Thus, the gap size in the resulting fragments was zero. Due to reduced complexity (see Section 2.3), the second tracking iteration can allow bigger gap sizes. Figure 3.1 shows a histogram depicting the sizes of the gaps contained in the truth tracks from dataset one. 90 % of them are smaller than 14 frames. So, if the classifier is allowed to only put together fragments with a gap size lower or equal to 14, it can merge 90 % of the fragments correctly. Larger gap sizes are be needed to merge more fragments. To cover 95 % of the gaps, the gap size should be 24, for 99 % already 90. To avoid slowing the tracking down, a gap size of 14 was chosen for the second tracking iteration. All larger gaps can be dealt with in further iterations of the tracking when complexity is again decreased by reducing the number of fragments. Now that the gap size is defined, the training fragments need to be generated.

A possible solution to obtain fragments for training is to generate *truth fragments* from the detections of the truth data. This means, merging detections from neighboring frames by their truth id: Detections that have the same truth id and that are located in adjacent frames form a fragment. These fragments could then be used for learning. This approach already works quite well, but it does not generate enough data, especially not for the positive class. In the truth dataset one, there are 5871 negative and 498 positive examples. In truth dataset two there are even fewer, 2377 examples for the negative class and only 204 for the positive class.

In machine learning, a larger training set can lead to better classification results [22]. According to Rajaraman [33], using more training data can even lead to more sophisticated results than implementing better algorithms. However, not only quantity, but also quality and diversity in the training data is required. Weiss and Provost [40] claim that only datasets that characterize all data are suitable for optimal training. The *truth fragments* are, hence, not suitable as they contain few examples for the positive

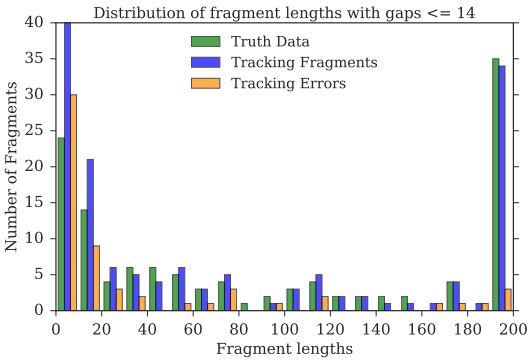


Figure 3.2: Length of fragments merged by the classifier and truth fragments. The tracking error indicates how many of the merged fragments contain inserts, deletes or are not merged correctly. The classifier was trained on dataset one and applied to dataset two.

class and few short fragments. Thus, the classifier cannot learn how to link short fragments correctly. Figure 3.2 depicts this problem: in the fragments merged by the classifier, there are many more short ones than in reality. The tracking error rate within them is very high.

To address this issue, a solution that can generate more training data, especially more short fragments, has been implemented. It is based on the truth tracks from both truth datasets. The truth tracks can contain gaps that are greater than 14 frames. Those gaps are not allowed in this tracking iteration. Therefore, the tracks are, first of all, split at gaps greater than 14 frames. The resulting tracks only contain gaps of the size that is allowed for this tracking iteration. Training fragments are then generated by splitting the tracks at every possible detection. Both halves of the tracks that result from each of these splits are added to the training data for class one. The examples for the negative class are generated by adding fragments from different tracks to the training fragments if there is no gap greater than 14 frames between them.

The short fragments for the positive class are generated the following way: for every two neighboring detections a and b in a track, the algorithm adds a and b to the training fragments as examples for the positive class. It does not perform this step for tracks of the length two, as for them, a and b are already added to the training data by the regular split. For every three neighboring detections a, b, and c in a track, it adds ab and c, and a and bc as training fragments with the positive class label. This step is not performed for tracks with a length of three as in these, ab and c, and a and bc are already added to the training data when splitting the track at every possible detection. Figure 3.3 visualizes the splitting process and the resulting training fragments.

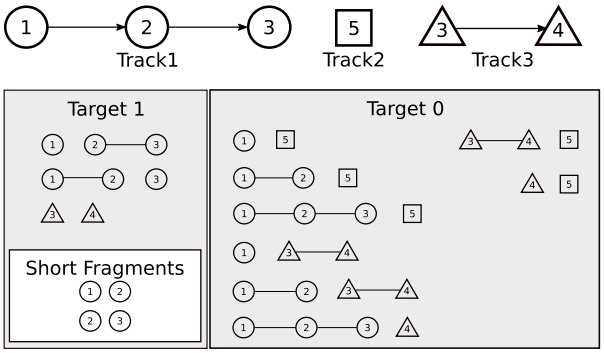


Figure 3.3: A minimalistic example of the training data generation. Tracks 1, 2, and 3 are given. The splitting algorithm generates the depicted training data from them. The training data with target 1 consists of fragments that should be merged, target 0 indicates that the fragments do not belong together. The short fragments are generated to train the classifier to merge short fragments better. For track 1, the fragments 12–3 and 1–23 are not added as short fragments, as they are already added as regular splits of the track. This is the same for track 3 (or any other track of length 2 or 3).

The algorithm produces a learning dataset characterized in Table 3.1. The class distribution corresponds approximately to the distribution in the real data. The size of the training data generated this way is about 170 times higher than with the *truth fragments*.

An evaluation of using generated data for training instead of the truth fragments points out, that the tracking improves at some important points. The number of detection *deletes*, i.e., detections that cannot be merged into their correct track decreases about 10 %. This is also reflected in the numbers of short fragments that are left after the tracking. Figure 3.4 shows the fragment length distribution of the fragments produced by the tracking with a classifier trained on truth fragments, and on generated data, and the true distribution (the distribution in the fragments of the truth tracks).

Table 3.1: The distribution of data in the generated learning dataset.

	Dataset from	
	18/09/2015	22/09/2015
Number of different truth fragments	215	123
Total number of generated training data points	1 021 848	698 003
Fragment pairs that should be merged (Target 1)	70 166	42 730
Fragment pairs that should not be merged (Target 0)	951 682	655 273
Percentage of data with label 1	7.4 %	6.5 %

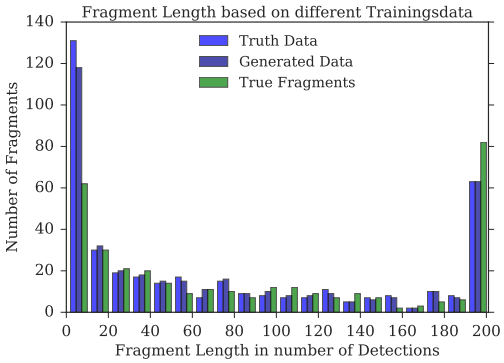


Figure 3.4: The classifier is trained on the truth fragments and the generated learning data. It is then used to predict the fragments for the other dataset as a sort of cross-validation. Comparison of fragment length between classifier on truth fragments and generated data. The classifier trained on generated data merges short fragments better. Having more short fragments than in the true fragments is not desirable. It leads to a decrease of the performance of the id assignment described in Section 3.2 and has a negative impact on the success of the analyses.

The tracking trained on generated data manages to merge more short fragments correctly. Section 3.2 will show that short fragments make the id assignment more difficult. Therefore, merging them correctly can already have a positive impact on the id assignment. However, for the long fragments, the training on generated data does not lead to an improvement. Both classifiers, the one trained on truth fragments, and the one trained on the generated data, only manage to merge about 80 % of the tracks of length 190-200. This needs to be improved in the following sections.

3.1.2 Classifier Selection

With the learning data prepared, the next task consists of choosing a well-performing classifier. Nowadays, there exist many classifiers that can perform binary classification tasks. However, some of them are more suitable for a particular task than others. Weiss and Provost [40] suggest testing a set of classifiers and selecting one based on an evaluation of their performance rather than based on personal preferences.

There exist multiple metrics to compare the performance of binary classifiers such as accuracy, recall, precision, F_1 score, ROC curves and ROC AUC score. They are based on the confusion matrix, with is shown in detail in Table 3.2. The following definitions for the performance metrics are taken from Fawcett [15].

Table 3.2: The confusion matrix with two rows and two columns that reports the number of false positives, false negatives, true positives, and true negatives. It allows visualization of the performance of an algorithm, typically a supervised learning.

		True Classification		
		Positive	Negative	Total
Predicted Value	Positive	True Positive (t_p)	False Positive (f_p)	$t_p + f_p$
	Negative	False Negative (f_n)	True Negative (t_n)	$f_n + t_n$
	Total	$t_p + t_n$	$f_p + f_n$	N

A metric widely used to evaluate the quality of a classification is the so-called *accuracy*. The accuracy describes the percentage of all correctly classified objects. The abbreviations used in the following equations are described in Table 3.2.

$$P\,(\text{Correctly Classified}) = \frac{t_p + t_n}{t_p + f_p + t_n + f_n}$$

However, due to the fact that the classes are very unbalanced, the accuracy is not a good metric to evaluate the classifier (see [40]). Even a classifier that classifies all fragment pairs to the negative class, and hence does not merge any fragment, could reach an accuracy of over 90 %.

A more suitable metric is the *recall* or *true positive rate*. The recall is defined as the percentage of positive objects that are correctly classified.

$$P\,(\text{Positive Prediction} \mid \text{Positive}) = \frac{t_p}{t_p + f_n}$$

A high recall rate is desirable in the tracking, as every false negative leads to a hole in a fragment. The resulting *deletes* cause information-loss in a track. If many short fragments cannot be merged, the id assignment gets more difficult and error-prone.

Analogous to the *true positive rate*, the *false positive rate* is defined as

$$P\,(\text{Positive Prediction} \mid \text{Negative}) = \frac{f_p}{f_p + t_n}.$$

Another important score is the *precision*. It is defined as the percentage of correctly as positive classified objects to all positive objects.

$$P(\text{Positive} \mid \text{Positive Prediction}) = \frac{t_p}{t_p + f_p}$$

A high precision rate might be even more desirable than a high recall rate. This is because if a fragment is merged to the wrong place, it produces *inserts*. Inserts cause even two problems. Firstly, an insert leads to information-loss as the wrongly inserted detections are missing in their correct track. Secondly, it might even show a wrong behavior in the track where it is inserted or lead to a wrong id assignment to this track.

The F_1 score is defined to combine precision and recall into one score. It is defined as

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

and it weights precision and recall the same. A different weighting based on the true costs of a false positive or false negative could be used, but these costs are hard to estimate.

Daskalaki, Kopanas, and Avouris [12] describe a metric that can be used without guessing the costs, the *ROC curves* (Receiver Operating Characteristics curves). ROC curves are 2-dimensional graphs that plot the true positive-rate on the y-axis and the false positive-rate on the x-axis. They thereby depict relative trade-offs between benefits (true positives) and costs (false positives) [15]. Figure 3.5 shows an example of a ROC graph for classifier evaluation. A classifier is better than another one if it leads to a higher true positive and a lower false positive rate. Informally spoken, in the ROC graph, a classifier performs better than another if it is situated northwest to it. In Figure 3.5, classifier A would perform better than classifier B.

As stated above, the ROC graphs are especially suitable for cases like the tracking, where the costs of classification errors might be different for both classes and where the class distribution is unbalanced [15]. However, the problem with ROC Curves is that, the quality of a classifier is visualized by a curve which makes comparisons difficult. Integrating the ROC curve yields a single comparable value called *AUC score* (area under the ROC Curve). Better classifiers are nearer to the left top point and have therefore a large integral. According to Bradley [8], the AUC score is very sensitive and offers a significant advantage compared to other metrics: it especially penalizes one-class-only or random classifications. This property makes the AUC Score a very suitable metric for the tracking.

Regardless, an evaluation on how well the classifier works for the actual tracking is important. This can be done by some custom metrics based on how many wrong

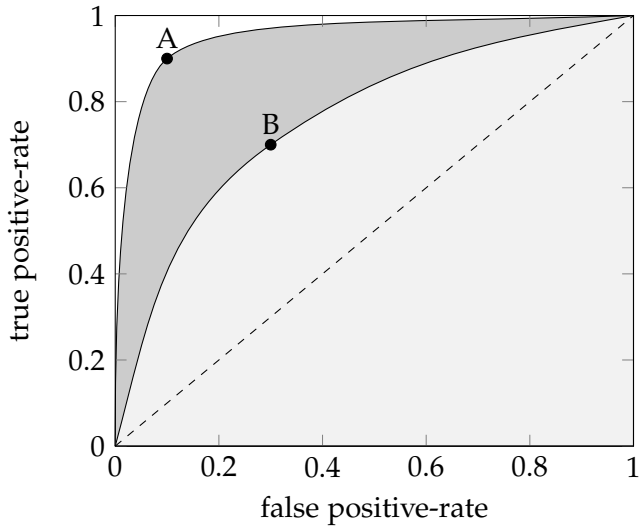


Figure 3.5: An example of a ROC graph: A and B represent the quality of two classifiers in ROC space. A performs better than B. The dashed line in the graph shows results of randomly guessing classifiers. If a classifier guesses the positive class in 50 % of the cases, it is likely to get 50 % of the positives right. However, as also 50 % of the negatives are classified positive, the false-positive rate will also be at about 50 %. The gray areas under the curves represent the AUC score belonging to the particular curve. Its value can be calculated by integrating the ROC curves.

detections or fragments are inserted into tracks (*inserts*) and how many detections or fragments could not be inserted to their right tracks (*deletes*).

The classifiers used for testing were taken from the Python scikit-learn library (often shorted as sklearn) [32] and offer a wide variety of different classification mechanisms. The following classifiers were evaluated: the Naive Bayes [31], a Support Vector Classification [20] with a radial basis function [9] kernel, a Support Vector Classification with a linear kernel, a limited and an unlimited Decision Tree [34], a Random Forest [27], an Ada and a Gradient Boost classifier [18, 19], and the XGBoost proposed by Chen and Guestrin [10]. To use the XGBoost together with the classifiers originating from the sklearn library the xgboost Python implementation [41] was used.

Figure 3.6 shows the qualities of the classifiers when being trained with a 10-fold cross-validation on the data of both datasets. The cross-validation is done by performing sklearn's stratified shuffle split on the generated learning fragment pairs. This randomly splits the dataset into 10 subsets containing approximately the same number of training fragment pairs together with their learning target. The class distribution in the subsets corresponds to the one in the original dataset. As the fragment pairs are chosen

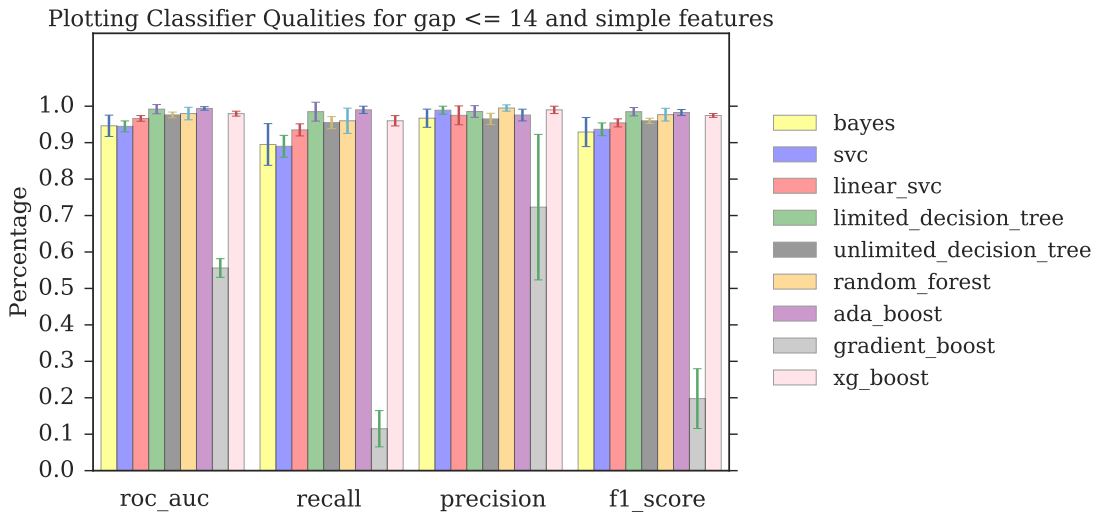


Figure 3.6: Scores obtained by the classifiers on both truth datasets with 10-fold cross validation. The mean value and the standard deviation are plotted.

randomly from the complete generated learning data, each subset contains fragments from each dataset. All of the classifiers were tested in their standard configuration with their default values. The gradient boost does not perform well with these settings. All of the other classifiers perform well, according to their ROC AUC score. Their performance concerning the recall varies more. Nevertheless, with these close results, it was difficult to choose one classifier.

However, when considering how well they perform to form fragments from detections, the classifiers differ more. Figure 3.7 shows the number of detection inserts, fragments with detections inserts, the number of detection deletes and fragments containing deletes. All four numbers should be low as stated above when describing the negative impacts of inserts and deletes. The figure shows that the classifiers with a higher ROC AUC score perform better than the ones with a lower one. The big differences in the numbers of detection deletes and inserts are explicable by the length of the fragments that could not be merged correctly. A classification error with a fragment that only contains a few detections is counted the same way as a classification error with a fragment that contains many detections.

In this regard, the random forest turns out to be the classifier which performs the tracking best. It produces very few inserts and the lowest number of detection deletes. Concerning the AUC and F_1 score, it also belongs to the best-performing classifiers. The random forest is a good choice as it is resistant against over-fitting and offers the advantage of allowing parallel computing [27], which might become important in the future of the BeesBook project (see Chapter 4).

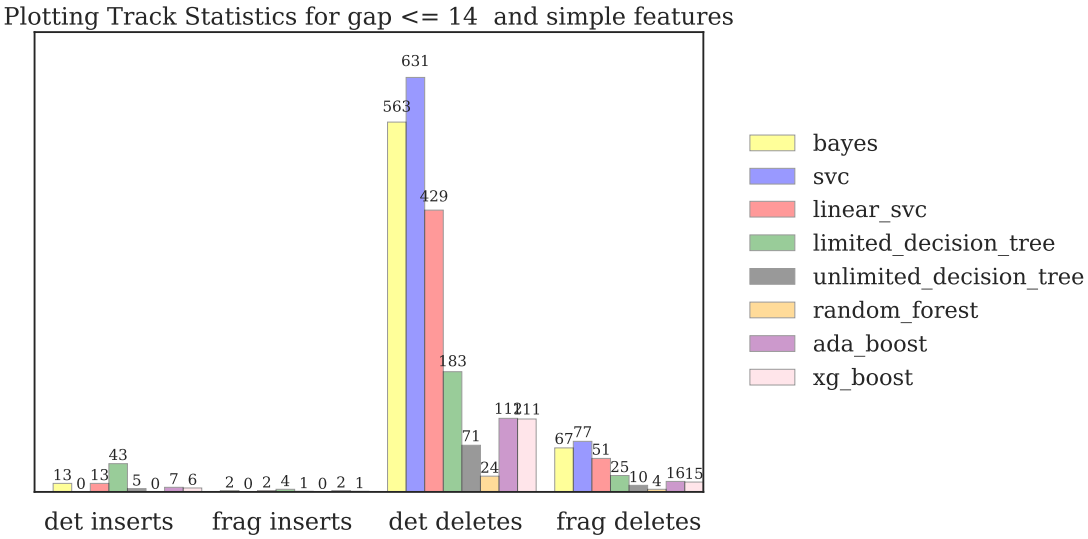


Figure 3.7: Tracking performance measured regarding inserts and deletes. The classifiers were trained on dataset one and predict dataset two. The number of detection inserts describes how many detections were inserted to the wrong track, or at the wrong position in their own track. The detection deletes refer to detections that are missing at their right position in their track. Fragment inserts and deletes represent the number of merged fragments that contain inserts or deletes. The gradient boosting has been left out in the diagram as it produces too many deletes to fit on the scale.

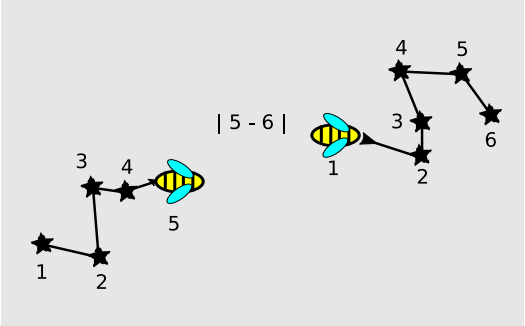
These findings agree with the finding of other research teams. They evaluated various classifiers for different classification and regression problems. They found out that in many cases, the random forest performs better than other classifiers [16].

3.1.3 Feature Extraction

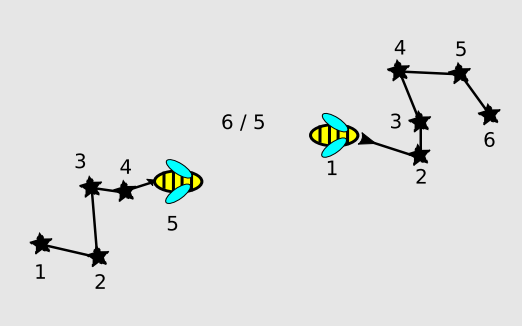
In this section, the features to support the classifier well in its classification task are developed. Bishop [7] describes the process of feature extraction as preprocessing input variables to new variables which make the solution of a given problem easier. For the tracking, the input variables are fragments consisting of bee detections and the variables of the data fields from the detections as described in Section 2.2. This information is barely useful to find out if some fragments form a track together or not. The given variables were, therefore, transformed into more significant features that could help to simplify the decision.

The following features were implemented additionally to the five described in Section 2.3 (`len1`, `len2`, `id_distance`, `distance`, and `time_diff`).

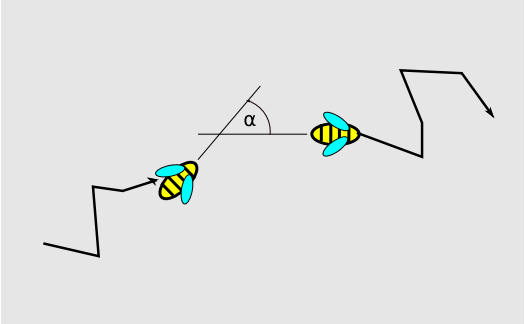
- `len_diff`: Absolute length difference between the two fragments (see Figure 3.8a), where length is the number of detections contained in a fragment.
- `len_quot`: Length of the second fragment divided by length of the first one (see Figure 3.8b).
- `x_rot_diff`: Absolute difference between the rotation on the x-axis between the last detection in the first fragment and the first detection in the second fragment.
- `y_rot_diff`, `z_rot_diff`: Same as `x_rot_diff` but on the y or z-axis (see Figure 3.8c).
- `x_rot_diff_per_time`: Divides `x_rot_diff` by the time elapsed between the two fragments to model the fact that a bee might move more if more time elapses.
- `y_rot_diff_per_time`, `z_rot_diff_per_time`: Same as `x_rot_diff_per_time` but on the y or z-axis (see Figure 3.8d).



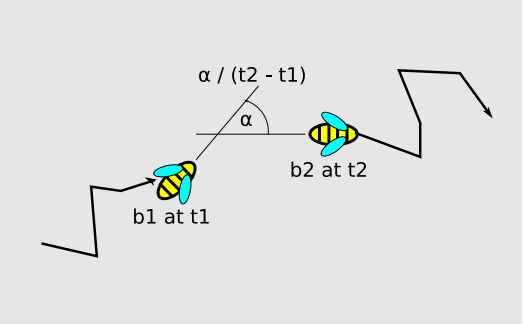
(a) `len_diff`



(b) `len_quot`



(c) `z_rot_diff`



(d) `z_rot_diff_per_time`

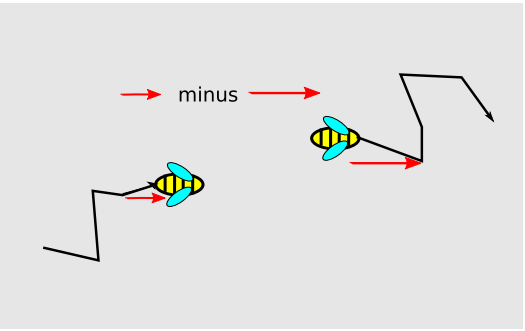
- `x_vector_diff`: Calculates the difference on the x-axis between two vectors given by the locations of the last (or first) two detections of the first (or second) fragment. As the two detections in both fragments might be separated by a different number of gaps in further tracking iterations, the vectors are first normalized by the time elapsed between the two detections (Figure 3.8e does not display the normalization but only the vector difference on the given axis).

- `y_vector_diff`: Same as `x_vector_diff` but on the y-axis (see Figure 3.8f).

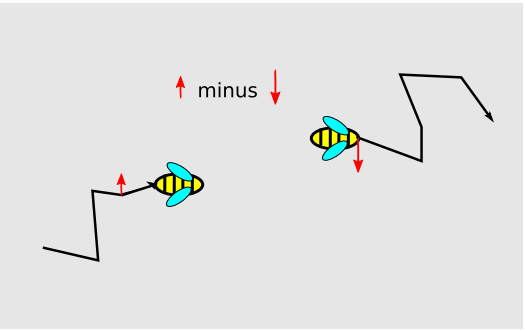
- `xy_vector_len_diff`: Similar to `x_vector_diff` but calculates the difference of the vector lengths instead of the vectors themselves (see Figure 3.8g).

- `xy_vector_angle`: Calculates the vectors as in `xy_vector_len_diff` and then returns the angle between them (see Figure 3.8h). It is normalized to $(-180°, 180°]$.

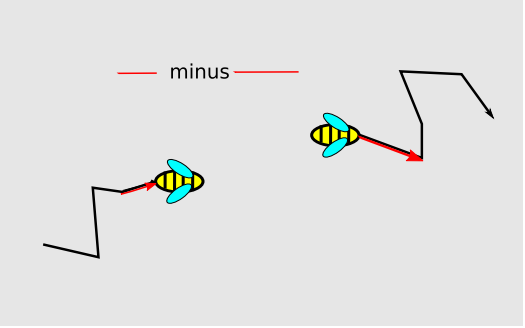
- `weighted_window_x_diff`: Similar to the `x_vector_diff`, but, if existent, not only the last (first) vector is included, but the last (or first) three. The vectors are then weighted according to their closeness to the end (or beginning) of the first (or second) fragment. The closer they are to the gap between the fragments, the higher they count (factors 1, 2, and 4).



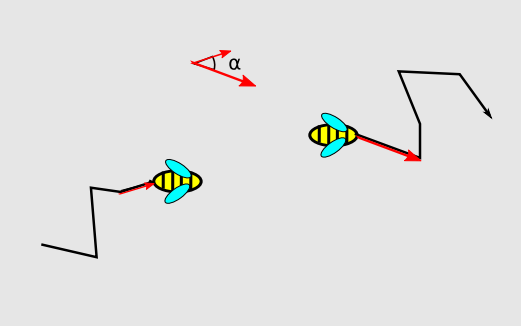
(e) `x_vector_diff`



(f) `y_vector_diff`



(g) `xy_vector_len_diff`



(h) `xy_vector_angle`

- `weighted_window_y_diff`: Same as `weighted_window_x_diff` but on the y-axis.

- `weighted_window_xy_len_diff`: Same as `weighted_window_x_diff` but concerning both axes.

- `gap_speed`: Euclidean distance between last detection of the first fragment and first detection of the second one, divided by the time elapsed between these two detections (see Figure 3.8i).

- `movement_area`: The area A covered by a fragment, defined as

$$A := \sqrt{(x_{\max} - x_{\min})^2 + (y_{\max} - y_{\min})^2}, \tag{3.1}$$

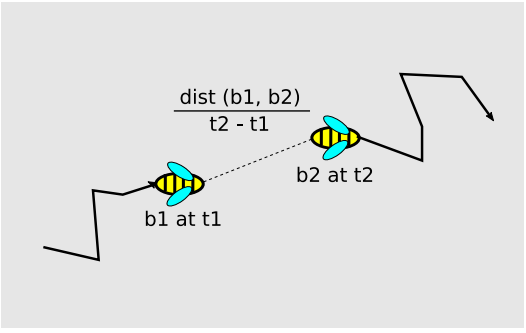
  where $x_{\max}$ and $x_{\min}$ are the largest and smallest x-coordinate of any detection in the fragment, and $y_{\max}$ and $y_{\min}$ the largest or smallest y-coordinates(see Figure 3.13c). To combine the two fragments' movement areas to one feature, the areas is normalized by the time elapsed during the fragments. Then, the absolute difference between the two areas is returned. This step is not represented in the picture, as a normalization by time is difficult to visualize.

- `confidence`: Let $T = d_1, d_2, \ldots, d_n$ be a track with n detections. A detection d_i consists (among others) of a twelve bit id, that is represented by twelve percentages indicating the certainty that is bit is set. The id of a detection can then be described as $d_i \in [0,1]^{12}$. The confidence c is defined as

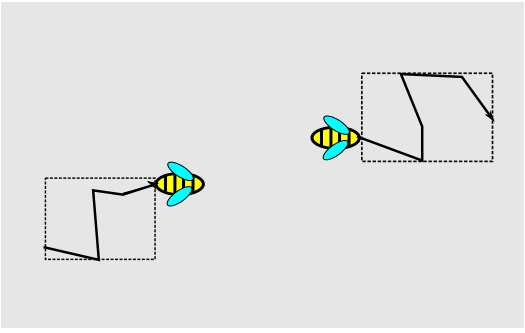
$$c := \operatorname*{mean}_{d \in T} \min_{i \in \{1,\ldots,12\}} 2 \cdot \left| \frac{1}{2} - d[i] \right|. \tag{3.2}$$

  This means, in all detections, the reliability of the weakest bit in the id is computed. The mean of this reliability over all detections is computed. The feature returns the absolute difference of the confidences of both fragments.

- `window_confidence`: Only the confidence of the last two or first two detections in the fragments is considered.



(i) `gap_speed`



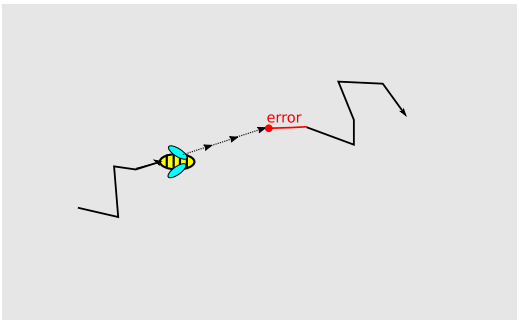
(j) `movement_area`

- `forward_error`: Calculates the vector given by the locations of the last two detections in the first fragment. It then returns the distance between the location where the bee would be expected by the time the second fragment starts (if she kept on moving exactly the same) and the real location of the first detection in the second fragment (see Figure 3.8k).

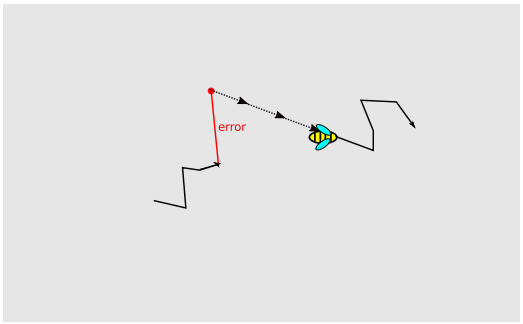
- `backward_error`: Similar to `forward_error`, but considers the first two detections of fragment two and calculates the error between the location where the bee should have been by the time fragment one ended and the actual location of the last detection in fragment one (see Figure 3.8l).

- `z_orientation_forward_error`: Similar to the `forward_error` but instead of calculating the error between a predicted and actual location, the difference between expected and real rotation are calculated. Therefore, the rotation between the last two detections of the first fragment is calculated and repeated until the beginning of the second fragment. The difference between the estimated and real rotation is returned. It is normalized to $(-180°, 180°]$ (see Figure 3.8m).

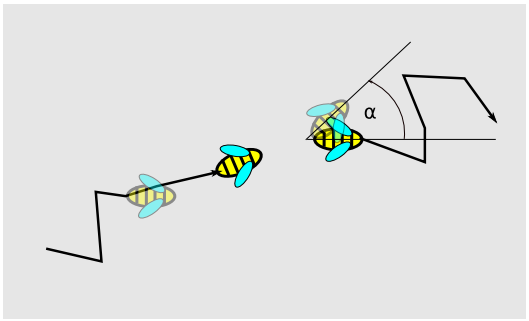
- `z_orientation_backward_error`: Similar to the `backward_error` and `z_orientation_forward_error` (see Figure 3.8n).



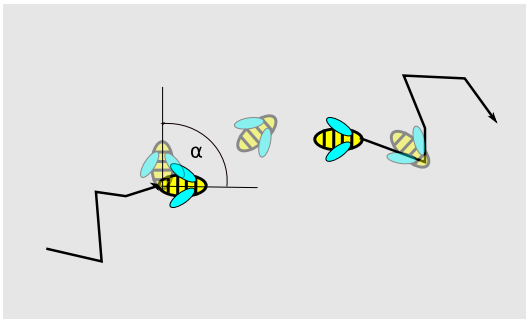
(k) `forward_error`



(l) `backward_error`



(m) `z_orientation_forward_error`



(n) `z_orientation_backward_error:`

Figure 3.8: Visualization of the complex features

- `speed_diff`: Absolute difference between the movement speed of both fragments. The movement speed is calculated by dividing the complete distance a bee covers within the fragment by the time needed.

- `len1_is_one`, `len2_is_one`: A boolean features that indicates whether fragment one (or two) only contains one detection. It was developed for some features that do not apply to fragments containing less than two detections. For example, all the vector features, together with the forward and backward error, rely on vectors specified through the location of two detections. In fragments, consisting of one detection only, it is thus not possible to calculate a value. In such cases, a default value which is not included in the co-domain of the feature is returned. For example, the forward and backward error, which normally return distances (positive values) return -1 in the case of the feature not being applicable. In combination with the boolean feature, the classifier should be able to learn the pattern of some features not being applicable and hence not relevant in some cases. The boolean features might be redundant due to the `len1` or `len2` feature or the `len_quot` or `len_diff`. This will be examined further in the next section.

It is important to notice that some features can be calculated more efficiently than others. For example, the `len1` or `len2` features only determine the length of a list, whereas features like the `speed_diff` need to iterate over the whole fragment and do calculations for every detection. This might slow the tracking down dramatically. Nevertheless, those features will not be penalized during the feature selection described in the following section, as this disadvantage could be limited. The feature's value might, for example, be saved together with the current number of detections in the meta data of a fragment to avoid calculating the same value again and again. This way, the saved value can be used until other detections are added to the fragment in which case the value needs to be updated. Such an approach would consume more memory but less computation time.

The feature set implemented in this section covers key aspects according to which fragments could be merged. In the following section, a feature selection is conducted to evaluate which subset of these features models the bees' behavior best and obtains the best tracking results.

3.1.4 Feature Selection

The search of a good feature subset is substantial to a well-performing tracking. However, a good performance does not only include producing good tracks but also running in a reasonable time. Kohavi and John [24] state that training on a smaller feature set is much faster than on a bigger one. They, therefore, propose penalizing bigger feature sets in the feature selection process. In the tracking, the number of features can not only

have a negative impact on the training times but also on the run-time of the tracking itself, as the features need to be calculated on the fly. This is because many of our features depend on variables of both fragments, so they cannot be pre-calculated. For this performance reason, the feature selection tries to find a minimal well-performing subset.

It is important to notice that in feature selection, often, there does not exist *one* optimal solution. This means that the optimal feature subset is not necessarily unique. Several solutions could perform equally well, based on the pre-defined metrics [24]. This is why one should approach the feature selection more as a question of finding features that are relevant for his problem. Kohavi and John [24] define a relevant feature as follows:

A feature X_i is relevant if and only if there exists some x_i with $P(X_i = x_i) > 0$ and some class y of the predictable class Y such that

$$P(Y = y|X_i = x_i) \neq P(Y = y).$$

Under this definition, X_i is relevant if knowing its value can change the prediction. Dash and Liu [11] describe the problem from a different perspective and consider feature selection as an elimination of irrelevant or redundant features. For the tracking, the first definition is more suitable, as the aim is to find the *minimal* well-performing feature subset.

Langley [26] divides feature selection into three different issues:

1. The starting point from within the set of features must be defined. It is possible to start without any feature, with all features, or somewhere in between.

2. The second issue concerns the organization of a search. One could perform an exhaustive search and try all different feature subsets. With n features, there are 2^n subsets. With the 31 tracking features, there would be $2^{31} = 2\,147\,483\,648$ different subsets. Training the classifier and cross-validating for each of these subsets would be computationally too expensive. Therefore, the search needs a strategy. One possible strategy consists of successively adding (or removing) features to the selected subset. No decision can be reversed and the resulting feature set cannot be changed again, it can only be extended (or reduced). Another strategy allows changing the current set again by adding and removing features before taking further decisions. This way, past decisions can be undone.

3. The evaluation strategy for the feature subsets to find out which one performs best.

There are two broad evaluation models for feature subsets proposed by John, Kohavi, and Pfleger [23], the *filter* and the *wrapper* methods. *Filter* methods rely on characteristics in the training dataset to find the most relevant features. They remove irrelevant features before the classification. Therefore, they are very efficient. However, the have several disadvantages [24]: Firstly, they do not take the bias of the classifier into account. The bias is the error resulting from wrong assumptions in the learning algorithm. Algorithms with a high bias might miss relevant relations between features and output targets. Secondly, they completely ignore the effect of the feature set on the performance of the classification. Therefore, they might fail completely in some scenarios. Finally, Guyon and Elisseeff [21] describe that filter methods tend to select weaker subsets than the *wrapper* methods. The *wrapper* methods run the classification on the data. Based on a given metric, they then evaluate the classification results to find the best performing feature set. A possible drawback of the wrapper methods is that the score for the metric to measure the quality of the classification needs to be calculated after every classification. This might lead to the fact that wrapper methods perform slightly slower than filter methods [26]. However, as they select stronger feature subsets, they are more suitable to select the feature subset for the improvement of the tracking.

To address the first two of the previously mentioned issues in feature selection, several standard feature selection approaches from sklearn were combined with a manual feature selection. They were applied to a dataset combining all of the data from both truth datasets constructed in Section 3.1.1. This avoids over-fitting the feature subset to a specific dataset. To do the cross-validation, the *stratified shuffle split* from sklearn was applied to the pairs of training fragments generated in Section 3.1.1 and their assigned target. This split divides the original dataset into subsets with approximately the same class distribution than the original dataset. The standard feature selection approaches from sklearn that were examined are the following:

Select-K-Best: The Select-*K*-Best method takes the dataset, a classifier, an evaluation metric, and a number of cross-validation sets. It returns the K features considered to work best given the evaluation metric. Here, the random forest was evaluated with the ROC AUC score (see Section 3.1.2) and a 3-fold cross validation was performed. In different runs, different values for K were specified. Table 3.3 shows the features selected for $K \in \{1, \ldots, 10\}$.

Recursive Feature Elimination: The recursive feature elimination starts with the full feature set and removes one feature per iteration. The decision which features to drop is taken by considering how they influence the classification results of the random forest. The advantage of starting the selection process with all features is that correlations between features can be detected. A specific feature might only perform well in combination with another one. These pairs cannot be found if features are added iteratively to the empty set, as the features alone might perform too poorly to be added. Table 3.4 presents the results of the recursive feature elimination. A comparison of the

Table 3.3: Results of the Select-K-Best approach for all $1 \leq K \leq 10$. As can be seen for larger K only new features are added.

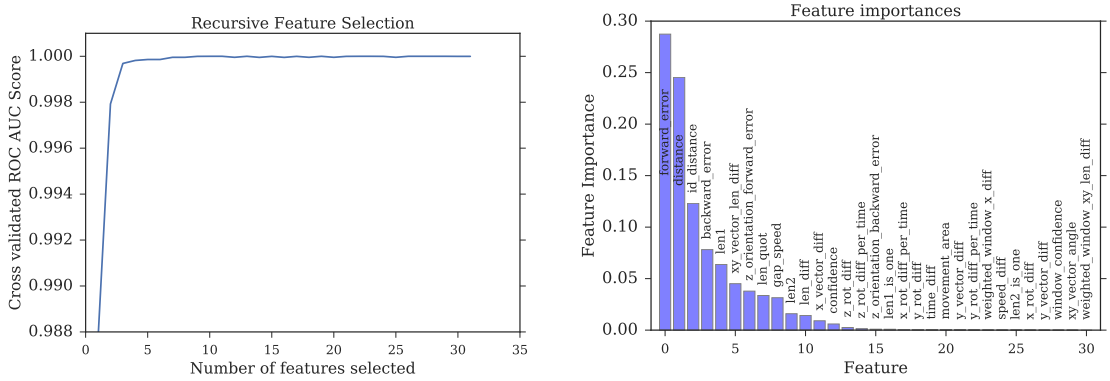
Feature	Number of features K									
	1	2	3	4	5	6	7	8	9	10
backward_error					✓	✓	✓	✓	✓	✓
confidence						✓	✓	✓	✓	✓
distance	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
forward_error				✓	✓	✓	✓	✓	✓	✓
gap_speed			✓	✓	✓	✓	✓	✓	✓	✓
id_distance		✓	✓	✓	✓	✓	✓	✓	✓	✓
len1									✓	✓
len_diff								✓	✓	✓
len_quot							✓	✓	✓	✓
speed_diff										✓

results with the ones of the Select-K-Best method shows that the best 5 features in both approaches are the same.

Recursive Feature Elimination with Cross-Validation: This approach adds a cross validation to the recursive feature elimination. The scikit-learn implementation also provides a method to calculate an optimal feature-subset. For the tracking, it returns an optimal

Table 3.4: Recursive feature elimination to determine feature subsets of size $1 \leq K \leq 10$. As can be seen, feature subsets can change between iterations.

Feature	Number of features K									
	1	2	3	4	5	6	7	8	9	10
backward_error				✓	✓	✓	✓	✓	✓	✓
distance		✓	✓	✓	✓	✓	✓	✓	✓	✓
forward_error	✓		✓	✓	✓	✓	✓	✓	✓	✓
gap_speed					✓	✓	✓	✓	✓	✓
id_distance		✓	✓	✓	✓	✓	✓	✓	✓	✓
len1									✓	✓
len2									✓	
len_diff							✓	✓	✓	✓
movement_area								✓		
xy_vector_len_diff										✓
z_orientation_forward_error										✓
z_rot_variance						✓		✓	✓	✓
z_rot_variance_per_time							✓			



(a) Recursive feature elimination with cross-validation: The x-axis represents that size of the selected feature subset, whereas on the y-axis the ROC AUC score of the corresponding feature subset is displayed.

(b) Feature importance in the random forest. The mean over all trees is considered.

Figure 3.9: Results of the Feature Selection Methods

feature subset which contains 16 features[1]. Figure 3.9a shows the AUC score for the strongest feature subsets of all sizes as determined by the recursive feature elimination with cross-validation. It seems as if already feature subsets of a size between 5 and 10 have a similarly high AUC score as the feature subset of size 16. It might, therefore, not be necessary to select such a large feature subset.

Feature Importance: The random forest provides a method to see the importance of the features used for classification. In sklearn, the feature importance is implemented as described by De'ath and Fabricius [13]. It uses the *gini criterion* that measures the quality of a split in a node defined by the total decrease in node impurity averaged over all trees of the random forest [37]. Figure 3.9b shows a plot of the mean feature importance over all trees. The features with the highest importance differ slightly from the top features selected by the other methods.

After performing these standard feature selection approaches, a manual selection has been implemented. It combines the top features selected by the methods described above to the best performing minimal feature subset.

The manual approach starts with four features that perform best in all methods: `id_distance`, `distance`, `forward_error`, and `backward_error`. Table 3.5 shows that those four already work a lot better than the five simple features used before and even better than the complete feature set of all 31 features.

[1]The features are `len1`, `len2`, `len_diff`, `len_quot`, `time_diff`, `distance`, `id_distance`, `z_rot_variance`, `x_rot_variance_per_time`, `z_rot_variance_per_time`, `y_vector_len_diff`, `gap_speed`, `movement_area`, `speed_diff`, `forward_error`, and `backward_error`.

Table 3.5: Comparison of the tracking results for different feature subsets. The training was conducted on dataset one and predicts dataset two. The other way around, the improvements are similarly high.

	Simple features	4 top features	All 31 features
Recall	0.8815	0.9199	0.8889
Precision	0.9620	0.9670	0.9771
F_1 score	0.9200	0.9429	0.9309
ROC AUC	0.9950	0.9965	0.9979
Tracks complete	60/76	64/76	60/76
Track fragments complete	82.11 %	84.55 %	80.49 %
Fragment inserts	2	1	3
Fragment deletes	49 (32.24 %)	40 (27.59 %)	50 (33.11 %)
Wrong fragments with $length = 1$	13	5	11
Wrong fragments $length \leq 5$	18	11	15

Starting with the four features, the manual approach adds one feature per iteration. To reduce computational complexity, only the features rated important by at least one selection method are considered. In every iteration, the feature that leads to the highest improvement in the classification is added and cannot be removed in further iterations. Features are added to the current set as long as adding a new feature leads to an improvement. The selection is stopped when adding any feature only leads to a performance decrease. With six features (third column), a local maximum score is reached (see Table 3.6). In the following iteration, no additional feature leads to a performance increase. The feature `gap_speed` causes the smallest decrease.

Table 3.6: Tracking results with the given feature subsets during the manual feature selection. The training was conducted on dataset one and predicts dataset two. The other way around, the improvements are similarly high.

	4 top features	Previous plus `confidence`	Previous plus `z_rot_diff`	Previous plus `gap_speed`
Recall	0.9199	0.9340	0.9375	0.9340
Precision	0.9670	0.9642	0.9747	0.9607
F_1 score	0.9429	0.9489	0.9558	0.9472
ROC AUC	0.9965	0.9966	0.9966	0.9986
Tracks complete	64/76	66/76	67/76	65/76
Track fragments complete	84.55 %	86.99 %	89.43 %	83.74 %
Fragment inserts	1	1	0	4
Fragment deletes	40	33	32	35
Wrong fragments $length = 1$	5	3	3	4

The feature selection stops at this point. The features `id_distance`, `distance`, `forward_error`, `backward_error`, `confidence`, and `z_rot_diff` form the minimal well-performing feature subset. It is important to notice that this special feature subset might only work so well with our selected random forest. If the classifier changes, another minimal feature subset might perform better because a given feature subsets supports different classifiers better than others [24].

3.1.5 Hyperparameter Tuning

Now, that the feature subset for training is chosen, the only step left to the model selection is the fine-tuning of the classifier. In machine learning, the classification algorithms can be configured before training. The configuration variables are called *hyperparameters* [4] and they differ from algorithm to algorithm. By tuning these hyperparameters, the results produced by a classifier can be immensely improved. Such an improvement can be even more significant than the whole modeling process or the feature engineering. A hyperparameter tuning should be the formal outer loop to a learning process [2]. This means that it should be performed after specifying the classifier and the feature set.

According to the sklearn documentation, the random forest has the following hyperparameters [1] that can be used to improve the classification. Other hyperparameters exist, but they serve only to make the classification easier (faster) or the results reproducible. Therefore, they do not need a fine-tuning.

- `bootstrap`: boolean variable that specifies whether bootstrap samples are used when building trees.
- `class_weight`: weights for both classes. It can be used to enforce an estimator to learn with attention to some class and thereby deal with unbalanced classes.
- `criterion`: function to measure the quality of a split.
- `max_depth`: the maximum depth of the trees.
- `max_features`: the number of features to consider when looking for the best split.
- `max_leaf_nodes`: maximal number of leaf nodes.
- `min_impurity_split`: threshold for early stopping in tree growth: a node will split if its impurity is above the threshold, otherwise it is a leaf.

- `min_samples_leaf`: the minimum number of samples required to be at a leaf node.
- `min_samples_split`: the minimum number of samples required to split an internal node.
- `min_weight_fraction_leaf`: the minimum weighted fraction of all the input samples required to be at a leaf node.
- `n_estimators`: the number of trees in the forest.

There exist several ways for tuning hyperparameters. One could perform a manual or grid search to get closer to the ideal combination of configuration parameters. In a classifier with K configuration variables, a set of values for each variable $\left(L^{(1)}...L^{(K)}\right)$ needs to be chosen. The set of trials is then formed by assembling every possible combination of values. This leads to a number of trials of

$$S = \prod_{k=1}^{K} \left| L^{(k)} \right| .$$

The number of configurations grows exponentially with the number of hyperparameters (curse of dimensionality [3]). Another drawback for the grid search is that one needs to identify the regions of interest manually. This task requires a good intuition for choosing the original set.

A better solution could be a random search. It has all the practical advantages of grid search (conceptual simplicity, ease of implementation, trivial parallelism) and is more efficient in high-dimensional search spaces [3]. A random search relies on randomly chosen trials instead of manually approximated ones.

For the tracking, the best hyperparameters for the random forest were calculated with the the hyperopt-sklearn library [25]. It used such a random search approach to calculate an optimal set of hyperparameters for a given classifier. Using the library instead of a manual approach prevents from having a human bias in the selection of the set of trials. Furthermore, it is much more efficient as its implementation is based on hyperopt [4], where efficiency is obtained by setting the search space with random variables and leaving out specific combinations of hyperparameters based on some heuristics.

Three different runs of the hyperopt-sklearn were carried out on the data of both datasets and obtained the three different sets of hyperparameters represented in Table 3.7. In each run, a different optimization metric (Accuracy, F_1 and AUC score) was specified, and the quality of the classification results was compared.

Table 3.7: Hyperparameters returned by hyperopt sklearn for the different optimization metrics.

Hyperparameter	Metric used for optimization: Accuracy	F_1 Score	AUC
bootstrap	True	False	False
class_weight	None	None	None
criterion	gini	gini	gini
max_depth	None	None	None
max_features	0.438	None	0.438
max_leaf_nodes	None	None	None
min_impurity_split	1e-07	1e-07	1e-07
min_samples_leaf	1	7	4
min_samples_split	2	2	2
min_weight_fraction_leaf	0.0	0.0	0.0
n_estimators	323	21	785

It turned out that the hyperparameters obtained by optimizing the AUC score led to the best tracking results in general. The parameters that optimized the accuracy were slightly worse and the ones for the F_1 score performed even worse to merge and complete tracks than the default parameters. A large drawback with the parameters optimized for the AUC score is the high number of estimators used for the prediction. Table 3.7 shows that the number of estimators in the hyperparameters optimized for the F_1 score is only 21 whereas it is 785 for the AUC score. Of course, a large number of trees (estimators) leads to more stable results in classifications with a random forest [27], but they also slow down the classification. Benchmarking revealed that a training of the classifier with the AUC hyperparameters took 1:57:32 hours, the prediction of one minute truth data 43 seconds. In comparison, the training with the accuracy parameters took 31:42 minutes and the prediction of one minute truth data less than 30 seconds. Test have been conducted to find out, if the quality of the tracking stays the same when reducing the number of estimators and keep the other parameters returned by the optimization for the AUC score. It turned out that performance decreases when reducing the number of estimators. Even with more than 500 estimators, the tracking performs worse that with the hyperparameters optimized for accuracy. As long as the tracking is performed by a single machine, it might, therefore, be acceptable using the hyperparameters optimized for accuracy, even though they perform slightly worse than the others, to find a good trade-off between quality of the prediction and time consumption.

With the hyperparameter tuning, the model selection process for the tracking improvement is completed. Before turning to the id assignment, it is time for an evaluation of all tracking improvements as a whole. Therefore, the tracking results obtained by

Table 3.8: Summary of the tracking improvement. The column Default represents the results of the tracking as currently integrated into the framework. The columns named Now refer to the performance with all the improvements proposed in the previous sections.

	Truth dataset used for training and for prediction			
	Trained on one predicts two		Trained on two predicts one	
	Default	Now	Default	Now
Recall	0.8537	0.9443	0.9500	0.9644
Precision	0.9570	0.9855	0.9870	0.9819
F_1 Score	0.9024	0.9644	0.9682	0.9731
ROC AUC	0.9977	0.9954	0.9995	0.9960
Tracks complete	59/76	69/76	109/123	112/123
Track fragments complete	78.05 %	91.06 %	87.91 %	87.91 %
Fragment inserts	2	0	1	3
Fragment deletes	64 (39.51 %)	25 (18.25 %)	52 (21.49 %)	40 (17.24 %)
Wrong fragments $length \leq 5$	25	5	21	11

the default solution that is currently integrated into the framework was compared to the results of the solution developed in the previous sections. Table 3.8 shows that the tracking improved in the important aspects. The new solution produces fewer inserts in dataset two and fewer deletes in both datasets. It furthermore leads to less short fragments that could not be merged correctly. Additionally, more tracks than before can be completed. All in all, this result suggests that the tracking improvements were a success.

3.2 Id Assignment

The currently used median id assignment as described in Section 2.2 does not assign the right ids to all fragments. Figure 3.10 shows that in about 10 % of the cases, the median id assignment even fails to assign the correct id to the manually labeled truth tracks. The success rate is even lower for the fragments obtained by the tracking. Therefore, the factors that cause the wrong id assignment need to be identified. Based on this knowledge, a more complex id assignment can then be implemented.

First of all, it is important to understand the kind of error that happens in the id assignment. To do so, the number of incorrect bits in the wrongly assigned median ids was examined. Figure 3.11a visualizes the results and reveals that there are many track ids with only one or two flipped bits, in other tracks, however, up to seven bits are flipped. Even a more complex id assignment might have difficulties correcting

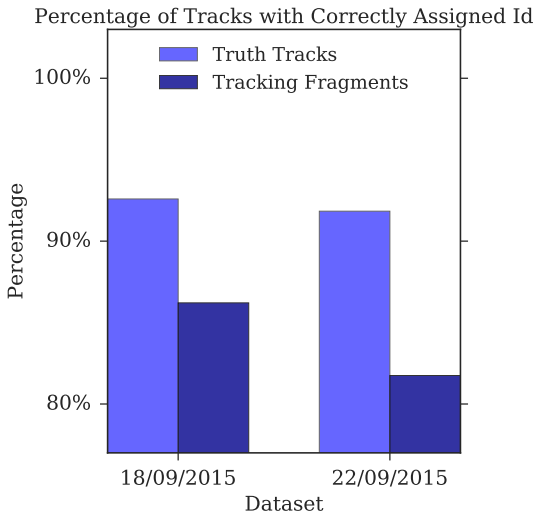
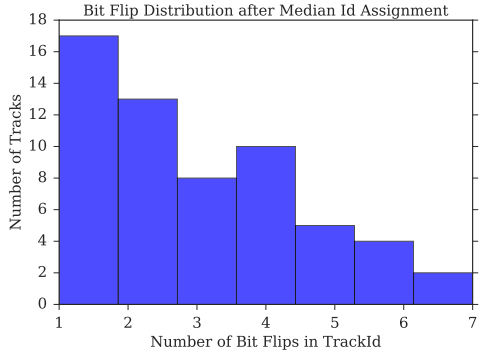
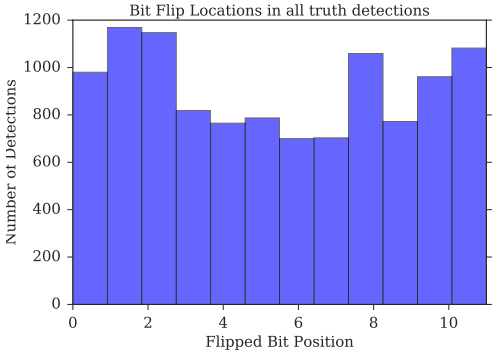


Figure 3.10: Quality of the median id assignment on the manually labeled truth tracks and the fragments determined by the tracking for both truth datasets.

that many errors. Furthermore it was investigated if some of the bits on the tags are more likely to flip than others. Therefore the locations of all bit errors produced by the decoder on the detections from both truth datasets were plotted in Figure 3.11b. It shows that the bit errors are not distributed equally over all locations in the tag. Some bits seem to break more easily than others and should therefore only be set when the decoder is confident enough. Apart from analyzing the kind of error in the id assignment, it needs to be analyzed in which fragments these errors appear.



(a) Number of bit flips in each track with a wrong median id. They are counted by calculating the hamming distance between the bit string of the median id and the fragment's truth id.



(b) Bit flip locations. They indicate the index of the bits that are incorrectly decoded. Index 0 corresponds to the segment on the tag that decodes 2^0.

Figure 3.11: Analysis of the bit errors in the wrongly assigned ids of both datasets.

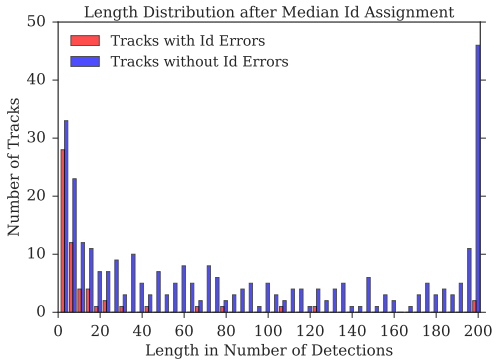


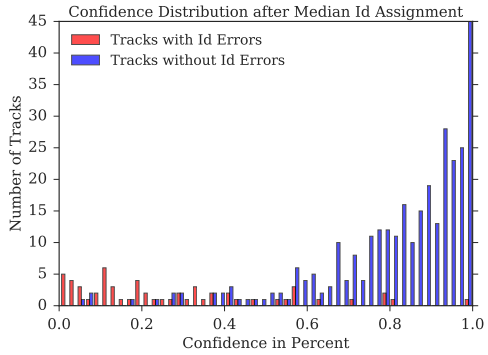
Figure 3.12: Comparison of the fragment lengths for fragments with correctly and wrongly assigned bits of both datasets. The fragment length is measured in number of detections; gaps are not counted.

The first characteristic of the fragments with wrongly assigned ids that was tested was their length. Figure 3.12 compares the length of fragments with correctly and wrongly assigned ids. It seems as if the wrong id assignments are more likely to appear in shorter fragments than in longer ones. This correlation might be because if a fragment only contains very few detections, its id cannot converge. In this case, a single outlier, a detection with a different id, might corrupt the whole track id.

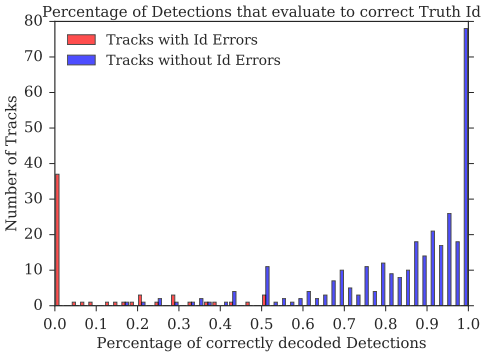
Another factor that influences the quality of the id assignment is the `confidence` of the detection ids returned by the decoder. The `confidence`, as defined in (3.2) on page 25, indicates how sure the decoder is about a decoded id. It is determined by the weakest bit. A low confidence implies that at least one of the bit probabilities is close to 0.5 in which case it is hard to decide whether a bit is set or not. As already one wrongly set bit leads to a different id, tracks with a low confidence are expected to fail more frequently in the id assignment than tracks with high confidence. Figure 3.13a confirms that most of the tracks with confidence under 0.6 evaluate to the wrong id with the median id assignment.

Furthermore, the impact of the percentage of detections with correctly decoded ids on the id assignment to their track was studied. A detection's id is decoded correctly, if the decoder returns the same id as the manually assigned truth id. Figure 3.13b shows that in most tracks with less than 50 % of the detections evaluating to the correct id, the assignment procedure gives a wrong id to the whole track.

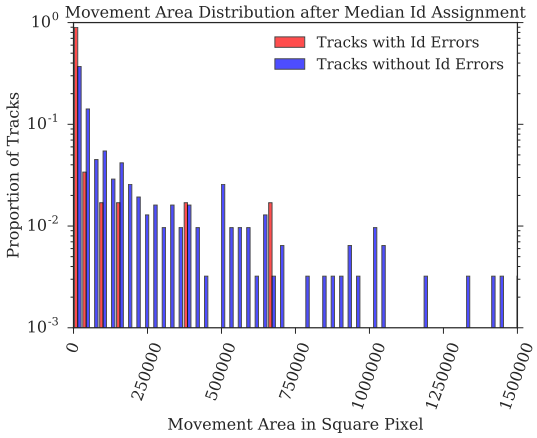
In addition, the movement areas, as defined in (3.1) on page 25, of tracks with wrongly and correctly assigned ids were compared. Figure 3.13c depicts the percentages of tracks covering movement areas of certain sizes. It points out that most of the tracks with a wrongly assigned id belong to bees that cover only small movement areas or do not move at all. This is explicable by the fact that if a bee changes her position, her



(a) Mean confidence of tracks with correctly and wrongly assigned ids. Most of the tracks with a very low confidence get a wrong id assigned.



(b) Percentage of correctly decoded bee ids in tracks with correct and wrong median ids. A correct decoded id corresponds in every 12 bits to the manually assigned truth id. Many of the tracks that get a wrong median id assigned only consist of detections with wrongly decoded ids.



(c) Movement area of fragments with correctly and wrongly assigned ids. A bee with a movement area close to zero does not actually move. Active bees have a bigger movement area. The scale in the diagram is logarithmic

Figure 3.13: Different factors that might influence the quality of the id assignment.

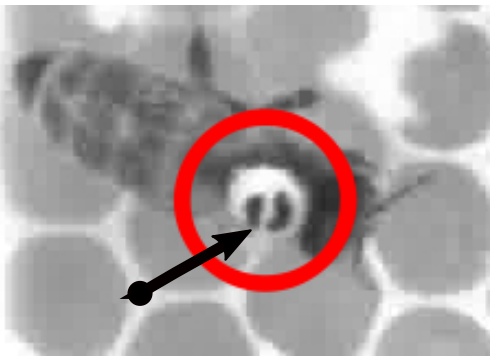


Figure 3.14: A bee with a tag that is hard to detect due to its rotation. The decoder returns id $[0,0,1,1,1,1,1,1,1,0,0]$ which corresponds to the decimal id 1020. However, the correct id is $[0,0,1,0,1,1,1,1,1,0,0]$ which is 1012. The value for the bit 2^3 is not detected correctly.

tag can be captured from different perspectives, but if she stays on the same spot, the decoder is likely to predict the same (wrong) id again.

Truth dataset one provides a good example of a bee's immobility leading to a wrong id assignment to her track. The bee is depicted in Figure 3.14. As she is clearly visible in all 201 frames and never covered, her track would be expected to receive the correct id, in this case 1012. However, the decoder returns 1020 with high confidence for nearly every detection. The reason is the tag's rotation which makes it hard to estimate whether the segment responsible to decode bit 2^3 is black or white. In this perspective, the black semi-circle could also just seem a bit longer than normal. As the bee does not change her position, the decoder has no chance of predicting its id better. In this example, the id assignment might not only fail because of the bees immobility but also due to the tag's rotation on the x-axis, the roll-axis, as shown Figure 2.4 (on page 8). All in all, it seems easier to decode tags of bees that lie flat on the comb than tags of bees that are rotated around the x or y-axis.

Figure 3.15 depicts the locations of the detections from tracks with wrong and correct ids. It is noticeable that most of the tracks with a wrongly assigned id are located close to the border of the comb. A possible reason is that the border of the comb is a wooden frame which is thicker than the comb itself. For this reason and because of the wide camera angle to the border the 3D model of the decoder network might not fit exactly to those tags and predict wrong ids. Thus, additional to the bee's rotation, her location is another factor that influences the id assignment.

Based on all those analyses, a more complex id assignment than the median has been implemented. The new method does not weight the id of every detection in a track

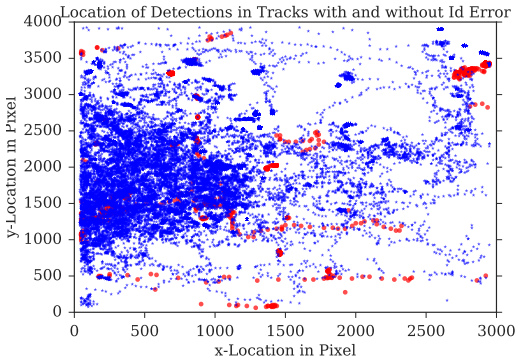


Figure 3.15: Location of the detections belonging to tracks with correct or wrong ids. The data displayed belongs to dataset one and represents all detections from 402 frames. The blue dots represent the detections of tracks with a correctly assigned id, the red dots of tracks with wrongly assigned ids.

equally. According to the factors that might influence the correct id assignment, some of the ids of detections are weighted higher, others lower. In the end, the mean of these weighted ids in the track is calculated, and bits with a probability greater than 50 % are considered to be set. In total, four different factors have been taken into account to calculate the weight of single detection ids. All weights have values in the range $[0, 1]$. Each of them was first evaluated separately. Afterward, they all were combined to one metric that forms the key part of the complex id assignment.

3.2.1 Euclidean Distance Weight

The first factor calculates the weight of a detection by it's Euclidean distance to its predecessor. It thereby represents the fact that more knowledge about the id represented by a tag is gained if the tag is captured from different perspectives. Analyses pointed out that the average distance covered by a bee between two pictures (0.3 seconds) is 20 pixel.

Let *dist* be the distance between two detection. The first detection of a track has factor $f_1(1)$=1, as the bee's appearance on the comb is considered as a significant movement. The first factor $f_1(i+1)$ for a detection d_{i+1} is calculated by

$$f_1(1) = 1$$
$$f_1(i+1) = \min\left(\frac{dist(d_i, d_{i+1})}{20}, 1\right).$$

The weighting by this model works quite well. Three of the tracks that were assigned a wrong id before do obtain the right id when weighting the detections by f_1 and calculating the mean id over the weighted detections. However two tracks that were assigned the correct id before, now have id errors. By analyzing those cases, it was found that in some tracks, a bee is visible for the first time when she stops working inside a comb cell. She then pulls her thorax out of the cell, and at the moment the picture is taken, she is still halfway inside. Due to this fact, a detection is created, but the decoder returns a wrong id, as some of the bits are not even visible. The tracking nevertheless manages to append this detection to the correct path. The problem with this is, that if the bee does not move for the rest of the track, only the wrong id is weighted with a high factor, and all other (correct) ids are barely considered. A similar thing happens when a bee covers another one by walking over it. The localizer might, in those cases, get the location of the tag wrong, so it is possible that the location of the tag jumps in the picture. Due to the occlusion caused by the other bee, the id is likely to be incorrect, but it would be weighted with a high factor as its position differs from the previous one. In this case, the id assignment might fail. To face those issues, the implementation of f_1 has been refined to find a more sophisticated weight. For each detection, not only its factor is considered, but the average of it and the previous detection's weight factor is calculated. This adds a sort of saturation to the model, so that outlier detections are no longer weighted that high. The idea is visualized in Figure 3.16. This saturation furthermore models that if a bee does not change her position in one or two frames, the detections might still be useful to confirm the decoded ids. However, the longer the bee is immobile, the less benefit is taken from the detections. With this model, movements are also weighted higher only if they last for some frames. This prevents weighting outliers and wrongly localized detections too much. To sum up, the weight $w_1(i+1)$ of detection d_{i+1} is calculated by

$$w_1(1) = 1, \text{ and}$$
$$w_1(i+1) = \frac{(w_1(i) + f_1(i+1))}{2}.$$

3.2.2 Rotation Distance Weight

The second factor weights detections according to the rotation distance to their predecessors. If a detection's rotation differs from the rotation of its predecessor, the tag can be captured from a different angle and thus, the information gain is higher. To calculate the rotation distance, a rotation matrix Q_{i+1} [14] for a detection d_{i+1} and Q_i for it's predecessor d_i is calculated. The second factor for detection d_{i+1} can then be determined by

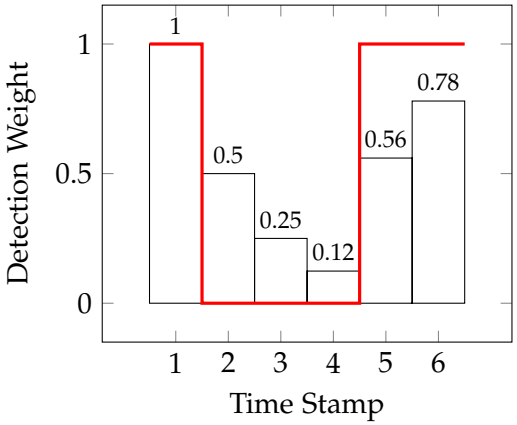


Figure 3.16: The red line plots the activity of a bee on a certain timestamp. 1 means that the bee takes a step and changes her position and 0 indicates that she stays immobile at the same spot. The bars represent the weight of the detection for the corresponding time step. The first detection is weighted with factor one, as the bee appeared on the hive. In the next detection, the bee has not changed her position. With our saturation model, its weight is calculated as the average of the previous weight (here 1), and its factor (here 0, as the bee does not move). This leads to a weight of $(1+0)/2 = 0.5$.

$$f_2(i+1) = 1 - \frac{1}{2}\left(\min\left(\operatorname{diag}\left(Q_i^T \cdot Q_{i+1}\right)\right) + 1\right).$$

The similarity of the two rotation matrices is given by their dot product. The values for each rotation axis are displayed on the diagonal of the square matrix. The axis with the least difference is considered. The resulting value is in the domain of $[-1,1]$. The factor should be in the range of $[0,1]$. Therefore, one is added and the value is divided by two. Now, a high similarity has value 1, a low one value 0. For the factor, this invert should be inverted, as detections with a similar rotation should be weighted lower. This is why the result is subtracted from one.

Similarly to the first weighting factor, the saturation model is used for the results. It considers the last rotation change together with the current one. This leads to a weighting $w_2(i+1)$ for detection d_{i+1} of

$$w_2(1) = 1, \text{ and}$$
$$w_2(i+1) = \frac{(w_2(i) + f_2(i+1))}{2}.$$

3.2.3 Position Weight

As described above, it might be easier to decode the id of bees that lie flat on the comb. So the third factor weight bees that are in the "normal" position higher than bees that are rotated around the x or y-axis. The rotation axes of the bee are represented in Figure 2.4 (on page 8). The rotation on the z axis does not have an impact on the readability of a tag, as it only indicates the orientation of the bee on the comb. To determine a bee's deviation from the "normal" flat position, first, the covariance matrix and the mean rotation over all detections are calculated. At this point, both truth datasets need to be analyzed separately. This is because, unfortunately, truth dataset one was created at a time when the rotation angles returned by the pipeline were not very accurate yet. The data from dataset one captured with camera zero and camera one also needs to be considered separately, as the rotations for camera one are rotated by π. This problem in the truth data does not have any impact on the previous factor that measures the rotation difference as the angles are at least consistent within one dataset. Calculating the covariance matrices and mean values for the independent datasets leads to the following matrices and vectors

$$\Sigma_{22,0} = \begin{pmatrix} 0.0135 & 0.0051 \\ 0.0051 & 0.0292 \end{pmatrix}, \text{ and } \mu_{22,0} = \begin{pmatrix} 0.06 & -0.43 \end{pmatrix},$$
$$\Sigma_{18,0} = \begin{pmatrix} 0.0021 & 0.0013 \\ 0.0013 & 0.0142 \end{pmatrix}, \text{ and } \mu_{18,0} = \begin{pmatrix} 0.06 & -0.43 \end{pmatrix},$$
$$\Sigma_{18,1} = \begin{pmatrix} 0.0166 & 0.0733 \\ 0.0733 & 4.2479 \end{pmatrix}, \text{ and } \mu_{18,1} = \begin{pmatrix} 0.24 & -0.41 \end{pmatrix}.$$

With these values, the multivariate normal distribution for a given detection's rotation concerning the covariance matrix and the mean value is calculated. This indicates her distance from the normal position. To make the factor compatible with the previous two weightings, every weighting greater than 1 is taken as 1. Let d_i be a detection and $r_i = (rot_x, rot_y)$ its rotation vector around x and y axis and Σ and μ the covariance matrix and mean vector corresponding to its dataset. Then, the third factor and weight with the multivariate normal distribution are calculated as

$$f_3(i) = \frac{1}{\sqrt{(2\pi)^2 det(\Sigma)}} \cdot e^{-\frac{1}{2}\cdot(r_i-\mu)^T\Sigma^{-1}(r_i-\mu)}$$
$$w_3(i) = \min\left(1, f_3(i)\right).$$

3.2.4 Confidence Weight

Finally, the fourth factor weights detections by their confidence. Detections with a low confidence might contain errors for at least one bit, so they should be weighted lower for the id assignment. In contrast, detections with high confidence that are less likely to contain errors should be weighted higher. The confidence for a single detection is similarly defined as for a track in (3.2) on page 25. The confidence produces values in the range $[0, 1]$. Therefore $w_4(i) = f_4(i)$. They are defined by

$$w_4(i) = f_4(i) = \min_{j \in \{1,\dots,12\}} 2 \cdot \left| \frac{1}{2} - d_i[j] \right|.$$

Figure 3.17 shows that the different weightings lead to results of different qualities. When weighting the detections only by confidence, the id assignment produces the lowest number of errors. An analysis of the truth ids of the tracks that can be improved by weighting the detections according to different factors shows that different factors can correct different track. Therefore, a complex weighting that combines all the factors might lead to the best results.

Such a combination could be implemented in the form of considering all factors independently by performing all id assignment methods one after the other. If all of them calculate the same id, this id is assigned to the track. If they disagree, either the id computed by the majority of the approaches (if a majority exists) can be assigned

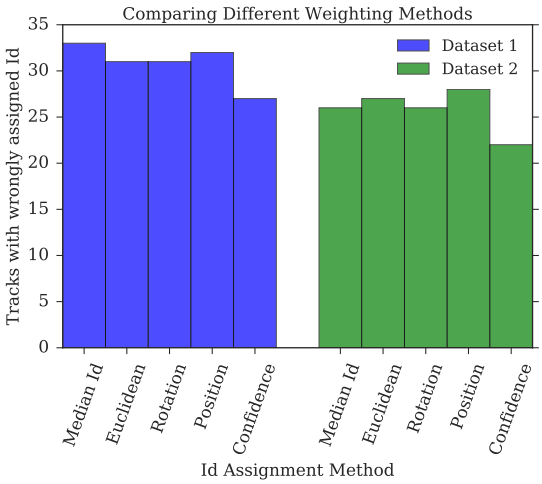


Figure 3.17: Comparison of the different id assignment methods. A method performs well if it leads to few tracks with wrongly assigned ids.

Table 3.9: Analyzing the combination of the different id assignment methods.

	Number of fragments in Dataset One	Number of fragments in Dataset Two
At least one method returns wrong id	36	31
No method found correct id	26	22
All methods agreed on same wrong id	17	6
Methods found different wrong ids	9	16
At least one method found correct id	10	9
Majority of methods found correct id	5	6

or another method to find the correct ids for those cases can be applied. Table 3.9 presents an analysis of the results of this combination method. In total, there are 67 of around 400 fragments for which at least one of the id assignment methods described above fails. For about 70 % of them, none of the methods returned a correct id. For 17 fragments, all methods even agree on the same wrong id. This behavior is undesirable as in this case, none of the methods can be used to correct the errors of another one. By assigning the id to the fragments with a majority vote, the ids of 11 fragments could be corrected. For 8 more, at least one approach found the right id. This corresponds to an improvement in the id assignment in all fragments of approximately 5 % compared to the median id assignment. However, 12 % of the fragments still obtain a wrong id.

There exist three options to deal with this inaccuracy of the id assignment that even appears within the complex methods. The first one would be to not assign any id to short fragments, fragments with low confidence, fragments that are located at the border of the comb or other fragments that might cause problems. However, as the id assignment works for some of those fragments, it would be a loss of information not assigning them any id at all. The second option could consist in assigning ids by leaving out some bits. Only bits that satisfy certain conditions would be set (like a high confidence) and the other ones would be left out. This would produce ids in the form of $[0, 0, 0, 0, 0, X, 0, 1, 0, 1, X, 1]$. With two empty bits, four alternative ids could belong to the fragment. For some analyses, this might be accurate enough. But the problem remains, that some long fragments, like the fragment belonging to the bee represented in Figure 3.14, have a high confidence for the wrong id. Therefore, this is also not a good option. The third possibility consists in deciding that an id assignment on detection base is not fine-grained enough for the given tracks. In this case, a bitwise id assignment over all detections of a track would have to be implemented. If every detection only contributed the correct bit(s), even fragments in which none of the detections is decoded correctly could receive the right id. This option will be discussed further in the following chapter.

4 Discussion

As presented in the previous chapter, the tracking has been improved substantially in all relevant measures. Thanks to the generated training data, the newly implemented behavior features, and the improved learning model, more fragments can be merged correctly such that about 90 % of the tracks can be completed. The computed fragments and tracks contain fewer inserts and deletes than before. Especially the short fragments are linked better than before and contain fewer errors. The results provide a solid basis for the analyses that will be conducted in the future.

The id assignment methods described in Section 3.2 provide better results than the median id assignment that was used before. Now, 19 of the 67 fragments for which the median id assignment fails can obtain their correct ids. This corresponds to a 5 % increase in the percentage of fragments with correct ids. However, for about 10 to 15 % of the fragments, still no method can determine the correct id. Therefore, the id assignment needs to be refined further. Weighting different ids in a track based on some characteristics of their detections does not seem to be fine-grained enough. Instead, different *bits* of each id in a track could be weighted differently. Bits with a high confidence or bits in certain constellations would count higher. The threshold that decides if a particular bit is set could also be defined bitwise. In the previous section, it was shown that some bits are assigned wrongly more often than others. For these bits, a different threshold could be set than for the other ones. An improvement of the id assignment could also be achieved by retraining the decoder. This training could be conducted on the tags that are, right now, decoded wrongly but with a high confidence, like the tag in Figure 3.14. In the previous section, it was shown that fragments are more likely to obtain a proper id if the percentage of detections with correctly decoded ids is high. Therefore, an increase in the accuracy of the decoder could improve the id assignment to the fragments.

In the future, the improved id assignment should furthermore return not only the estimated id, but also a score indicating the confidence of this id. Fragments or tracks with uncertain ids could then be excluded from the analyses as they might lead to wrong findings. The previous section showed that in some cases, all id assignment methods predict the same wrong id. Therefore, calculating the confidence score based on the number of methods that predict the same id does not seem to be a good solution. It might make more sense to calculate the confidence by some factors that are relevant

for the id assignment, e.g., fragment length, location on the comb, movement area, and confidence.

The newly implemented id assignment methods could also be used to improve the tracking itself. A very highly rated feature in the tracking is the `id_distance`. It computes how different the fragment ids are from each other. Currently, the fragment ids are calculated by the median id assignment. However, the previous chapter showed that other methods determine the fragment ids better. They could, thus, replace the median id assignment in the `id_distance`. This improved feature could then lead to better classification results.

As the second iteration of the tracking works well now, further iterations should be performed to merge the current fragments into longer ones. This has a positive impact on the id assignment that works better for longer fragments than for shorter ones and provides longer tracks for the analyses. The number of iterations that still needs to be performed depends on the quality of the id assignment. When most of the fragments obtain their right id, no more probabilistic tracking is needed, and fragments can be merged by comparing their ids. It is possible that some of the features optimized for a maximal gap size of 14 frames (4.66 seconds) might not have the same predictive power in further iterations with longer gaps anymore. This could be the case for the forward and backward error which could become meaningless when the gap between the two fragments is so long that the bee can change her movement direction multiple times. In this case, a different feature extraction and selection becomes necessary. The reason why the next iteration of the tracking has not been performed yet is the lack of truth data. In Section 3.1.2, a gap of 14 frames was selected that covered 90 % of the gaps contained in the truth data (see Figure 3.1 on page 14). The remaining 10 % are not sufficient to train and evaluate a classifier. Currently, an efficient way of generating truth data is implemented. It uses the tracks generated by the tracking itself. After a manual correction, these tracks and their manually assigned id are added as truth tracks. Based on this data, further experiments can be conducted.

Another important task to be tackled is the integration of the new model into the BeesBook system such that the improved tracks become accessible to the users. The tracking could furthermore be parallelized to improve its runtime. Currently, it needs 20–30 seconds on one core to track one minute of truth data. To track the data of a complete season, this might still be too slow.

Despite all the improvements that can be done in the future, the system is already in a state where the analyses can be started and communication of the bees can be investigated as specified at the beginning of this thesis.

Bibliography

[1] *3.2.4.3.1. Sklearn.Ensemble.RandomForestClassifier — Scikit-Learn 0.18.1 Documentation*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html (visited on 01/19/2017).

[2] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems*. 2011, pp. 2546–2554. URL: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization (visited on 01/06/2017).

[3] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13 (Feb 2012), pp. 281–305. URL: http://www.jmlr.org/papers/v13/bergstra12a.html (visited on 01/06/2017).

[4] James Bergstra, Dan Yamins, and David D. Cox. "Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms". In: *Proceedings of the 12th Python in Science Conference*. Citeseer, 2013, pp. 13–20. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.704.3494&rep=rep1&type=pdf (visited on 01/06/2017).

[5] *BioroboticsLab/Bb_binary*. URL: https://github.com/BioroboticsLab/bb_binary (visited on 01/11/2017).

[6] *BioroboticsLab/Bb_tracking*. URL: https://github.com/BioroboticsLab/bb_tracking (visited on 01/10/2017).

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. New York: Springer-Verlag, 2006. 738 pp.

[8] Andrew P. Bradley. "The Use of the Area under the ROC Curve in the Evaluation of Machine Learning Algorithms". In: *Pattern Recognition* 10.7 (1997), pp. 1145–1159.

[9] Martin D. Buhmann. "Radial Basis Functions". In: *Acta Numerica 2000* 9 (2000), pp. 1–38. URL: http://journals.cambridge.org/article_S0962492900000015 (visited on 02/17/2017).

[10] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *arXiv preprint arXiv:1603.02754* (2016). URL: http://arxiv.org/abs/1603.02754 (visited on 06/18/2016).

[11] Manoranjan Dash and Huan Liu. "Feature Selection for Classification". In: *Intelligent data analysis* 1.3 (1997), pp. 131–156. URL: http://content.iospress.com/articles/intelligent-data-analysis/ida1-3-02 (visited on 01/06/2017).

[12] Sophia Daskalaki, Ioannis Kopanas, and Nikolaos Avouris. "Evaluation of Classifiers for an Uneven Class Distribution Problem". In: *Applied Artificial Intelligence* 20.5 (June 2006), pp. 381–417. ISSN: 0883-9514, 1087-6545. DOI: 10.1080/08839510500313653. (Visited on 01/06/2017).

[13] Glenn De'ath and Katharina E. Fabricius. "Classification and Regression Trees: A Powerful yet Simple Technique for Ecological Data Analysis". In: *Ecology* 81.11 (2000), pp. 3178–3192. URL: http://onlinelibrary.wiley.com/doi/10.1890/0012-9658(2000)081%5B3178:CARTAP%5D2.0.CO;2/full (visited on 02/17/2017).

[14] *Drehmatrix*. In: *Wikipedia*. Page Version ID: 157447832. Aug. 28, 2016. URL: https://de.wikipedia.org/w/index.php?title=Drehmatrix&oldid=157447832 (visited on 01/10/2017).

[15] Tom Fawcett. "ROC Graphs: Notes and Practical Considerations for Data Mining Researchers". In: *HPL-2003-4* (2003). URL: http://www.hpl.hp.com/techreports/2003/HPL-2003-4.pdf (visited on 01/13/2017).

[16] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. "Do We Need Hundreds of Classifiers to Solve Real World Classification Problems". In: *J. Mach. Learn. Res* 15.1 (2014), pp. 3133–3181. URL: http://www.jmlr.org/papers/volume15/delgado14a/source/delgado14a.pdf (visited on 01/06/2017).

[17] Mark Fiala. "ARTag, a Fiducial Marker System Using Digital Techniques". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. IEEE, 2005, pp. 590–596.

[18] Yoav Freund and Robert E. Schapire. "A Short Introduction to Boosting". In: *Journal of Japanese Society for Artificial Intelligence* 14(05) (1999), pp. 771–780.

[19] Jerome H. Friedmann. "Stochastic Gradient Boosting". In: *Computational Statistics & Data Analysis* 38 ().

[20] Steve R. Gunn. *Support Vector Machines for Classification and Regression*. Technical. University Of Southhampton, Oct. 5, 1998.

[21] Isabelle Guyon and André Elisseeff. "An Introduction to Variable and Feature Selection". In: *Journal of Machine Learning Research 3* 3 (Mar. 2003), pp. 1157–1182.

[22] Alon Halevy, Peter Norvig, and Fernando Pereira. "The Unreasonable Effectiveness of Data". In: *IEEE Intelligent Systems* 24.2 (2009), pp. 8–12. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4804817 (visited on 01/06/2017).

[23] George H. John, Ron Kohavi, Karl Pfleger, et al. "Irrelevant Features and the Subset Selection Problem". In: *Machine Learning: Proceedings of the Eleventh International Conference*. 1994, pp. 121–129.

[24] Ron Kohavi and George H. John. "Wrappers for Feature Subset Selection". In: *Artificial Intelligence* 97 (1996), pp. 273–324.

[25] Brent Komer, James Bergstra, and Chris Eliasmith. "Hyperopt-Sklearn: Automatic Hyperparameter Configuration for Scikit-Learn". In: *ICML Workshop on AutoML*. 2014. URL: http://compneuro.uwaterloo.ca/files/publications/komer.2014b.pdf (visited on 01/06/2017).

[26] Pat Langley et al. "Selection of Relevant Features in Machine Learning". In: *Proceedings of the AAAI Fall Symposium on Relevance*. Vol. 184. 1994, pp. 245–271. URL: http://www.aaai.org/Papers/Symposia/Fall/1994/FS-94-02/FS94-02-034.pdf (visited on 01/06/2017).

[27] Andy Liaw and Matthew Wiener. "Classification and Regression by randomForest". In: *R news* 2.3 (2002), pp. 18–22. URL: ftp://131.252.97.79/Transfer/Treg/WFRE_Articles/Liaw_02_Classification%20and%20regression%20by%20randomForest.pdf (visited on 01/06/2017).

[28] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction". In: *International Conference on Artificial Neural Networks*. Springer, 2011, pp. 52–59. URL: http://link.springer.com/10.1007%2F978-3-642-21735-7_7 (visited on 02/15/2017).

[29] Danielle P. Mersch, Alessandro Crespi, and Laurent Keller. "Tracking Individuals Shows Spatial Fidelity Is a Key Regulator of Ant Social Organization". In: *Science* 340.6136 (2013), pp. 1090–1093.

[30] Jakob Mischek. "Probabilistisches Tracking von Bienenpfaden". Masterarbeit. Freie Universität Berlin, 2016.

[31] Kevin P. Murphy. "Naive Bayes Classifiers". In: *University of British Columbia* (2006). URL: http://www.ic.unicamp.br/~rocha/teaching/2011s1/mc906/aulas/naive-bayes.pdf (visited on 02/17/2017).

[32] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-Learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (Oct 2011), pp. 2825–2830. URL: http://www.jmlr.org/papers/v12/pedregosa11a.html (visited on 01/06/2017).

[33] Anand Rajaraman. *More Data Usually Beats Better Algorithms*. URL: http://anand.typepad.com/datawocky/2008/03/more-data-usual.html (visited on 01/10/2017).

[34] Lior Rokach and Oded Maimon. "Decision Trees". In: *Data Mining and Knowledge Discovery Handbook*. Springer, 2005, pp. 165–192. URL: http://link.springer.com/chapter/10.1007/0-387-25465-X_9 (visited on 02/17/2017).

[35] Benjamin Rosemann. "Ein Erweiterbares System Für Experimente Mit Multi-Target Tracking von Markierten Bienen". Masterarbeit. Freie Universität Berlin, 2017. in prep.

[36] *Sandstorm-Io/Capnproto*. URL: https://github.com/sandstorm-io/capnproto (visited on 01/11/2017).

[37] *Scikit Learn - How Are Feature_importances in RandomForestClassifier Determined? - Stack Overflow*. URL: http://stackoverflow.com/questions/15810339/how-are-feature-importances-in-randomforestclassifier-determined (visited on 02/17/2017).

[38] Thomas D. Seeley. *The Wisdom of the Hive: The Social Physiology of Honey Bee Colonies*. Cambridge, Mass: Harvard University Press, 1995. 295 pp. ISBN: 978-0-674-95376-5.

[39] Fernando Wario, Benjamin Wild, Margaret J. Couvillon, Raúl Rojas, and Tim Landgraf. "Automatic Methods for Long-Term Tracking and the Detection and Decoding of Communication Dances in Honeybees". In: *Frontiers in Ecology and Evolution* 3 (Sept. 25, 2015). ISSN: 2296-701X. DOI: 10.3389/fevo.2015.00103. (Visited on 01/10/2017).

[40] Gary M. Weiss and Foster Provost. "The Effect of Class Distribution on Classifier Learning: An Empirical Study". In: *Rutgers Univ* (2001). URL: ftp://ftp.cs.rutgers.edu/http/http/cs/pub/technical-reports/work/ml-tr-44.pdf (visited on 01/13/2017).

[41] *XGBoost Python Package — Xgboost 0.6 Documentation*. URL: http://xgboost.readthedocs.io/en/latest/python/index.html (visited on 01/19/2017).

[42] *Yaw, Pitch, and Roll Rotations*. URL: http://planning.cs.uiuc.edu/node102.html (visited on 01/10/2017).