LEARNING DESIRED BEHAVIOUR FOR CAUSALITY ANALYSIS

Hendrik Leidinger

Master's Thesis

Reactive Systems Group
Faculty of Natural Sciences and Technology I
Department of Computer Science
Saarland University

Supervisor:
Prof. Bernd Finkbeiner, Ph. D.

Advisor:
Maximilian Schwenger

Reviewers:
Prof. Bernd Finkbeiner, Ph. D.
Dr. Daniel Neider

Submitted: July 2019

Declaration of Authorship

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Datum/Date: _____

Unterschrift/Signature: _____

"If you torture the data long enough, it will confess."

- Ronald Coase

SAARLAND UNIVERSITY

Abstract

Reactive Systems Group
Department of Computer Science

Master of Science

by Hendrik Leidinger

Failure Analysis enjoys a high economic interest, as any kind of failure may result in incalculable risks and costs. It is concerned with the collection and analysis of data for detecting causes of failures within a system. A widespread technique is fault tree analysis, which enables a graphical representation of possible failures and their causes. However, fault trees still do not save the system designer the tedious work of searching for the actual cause. Most approaches require full manual analysis of a misbehavior based on logs and system designs. Previous approaches to the automation of such failure analyses have not yet been able to spread across the industry as they typically focus on very specific problems.

We present an approach to support the user in the search for the failure cause of a system run with the help of machine learning. The analysis is based on the log files generated by the monitored system. Our approach is now structured in three steps. First the log is scanned for segments where abnormal behavior occurs. A machine-learned classifier then determines the type of the anomaly. Finally, we generate a regressor for each anomaly type, which estimates the level of severity of the respective anomaly. If a system run fails, we search for combinations of anomalies whose summed severity exceed a certain threshold. These combinations are presented to the user as possible causes of failure. In concrete terms, we validate our approach on a quadcopter which is an unmanned aerial vehicle having gained increasing attention in the more recent past. We utilize a state-of-the-art simulation tool to generate data for training and evaluation.

Acknowledgments

This master's thesis would not have been possible without the moral support of certain people. First and foremost I would like to thank Lea Gröber who kept motivating me to continue and never got tired of proofreading.
Furthermore I would like to thank my advisor Maximilian Schwenger. The weekly discussions helped me to keep sight of the goal and also provided me with impulses for proceeding further. Of course I would like to thank him for proofreading as well.
Especially I would like to thank my parents who have always backed me up.
Last but not least I want to thank Professor Bernd Finkbeiner and Dr. Daniel Neider for reviewing my thesis.

Contents

Chapter 1

Introduction

Today, a variety of systems are characterized by a complex interaction of hardware and software components. The correct functioning of these systems, such as vehicles, airplanes or drones, is influenced by many external and internal conditions, making the troubleshooting process difficult. For example, a failed system run can be the result of a software bug or adverse environmental influences, which we call *faults* or *causes*. Understandably, there is a high economic and ethical interest in developing systems that are as error-free as possible, and to this end ever better methods are being developed for identifying the causes of failures.
The generic term for the collection and analysis of data to find or prevent a cause of failure within a system is called Failure Analysis. In this context, a failure denotes a state of the system that leads to the system no longer being able to perform its task successfully. Failure Analysis of a complex system, such as an autonomous drone, is a challenging task. This is not only due to the intrinsic complexity of such systems, but also to the fact that they react to and rely on their environment, which in turn influences the behavior of the system. For example, a drone might not reach its destination in time because it has to counteract a wind level that would otherwise cause it to lose course. In the following we present two approaches that support engineers during Failure Analysis.

Fault tree analysis [1] tries to capture all possible causes and effects in a tree-like structure with logical connections. The causes can then be found by traversing the tree from the failure event. However, creating a fault tree to a complex system is non trivial. The tree must be created manually and all possible sources of error must be identified in advance. There are approaches to generate fault trees automatically [2, 3], but these are very specific to underlying architectural languages, which describe the layout of a system. With our approach causes of errors are not only identified for an abstract system, as is the case with fault tree analysis, but are also specifically related to the system run.
Another method to support engineers during Failure Analysis is runtime verification. It is

a technique for the verification of a single run of a system by processing information about its states [4]. Runtime verification provides a more detailed insight into the behavior and occurring errors of the system in contrast to techniques like testing. This information can be valuable during the search for the possible failure causes. However, a manual search for the faults is still required. In addition, runtime verification generically shows all possible errors. Our approach strongly narrows down the number of possible causes and thus additionally eases the workload with the help of machine learning techniques. In the following we explain our approach on an abstract level.

Step 1 - Anomaly Detection

Based on the log files generated by the monitored system, we proceed as follows: In a first step, we generate sampling points from the log that describe the system flow. For each of these sampling points, we then find out whether the system behaves abnormally there. This is done with the help of a binary classifier for each sampling point, which determines whether the behavior at this point is abnormal or not.

Step 2 - Type Identification

In the second step, the focus is on identifying the type of anomaly based on the behavior of the system. The idea is that different sources of error cause specific behavior of the system, which in turn allows deductions about the anomalies. For example, drifting drones always behave differently, depending on which errors occur. In a gust of wind, for example, they try to correct the deviation by countersteering. In contrast, if the GPS data is incorrect, drones correct the trajectory, which is significantly different from the normal flight path. In other words, the task of the second step is to identify the type of anomaly, be it a gust of wind, incorrect GPS data, or even both. The goal is therefore to find out which types of anomalies occur for sets of subsequent sampling points and where the number of possible types is limited. This happens again with the help of machine learning. For step 2 we have to deal with a problem of classification with multiple labels. Thereby the classifier marks a sequence of sampling points with all anomaly types it exhibits.

Step 3 - Anomaly Scoring

Last, we determine which anomalies are at fault in the event of a system failure. For this purpose, we train a regressor for each type of anomaly to assess the severity of the anomaly. If an anomaly has a high negative impact on the system, it will receive a high

score. We adjust the overall magnitude of all anomaly types using the training data. Since we know which runs failed and which did not, we can ensure that the sums of the scores for the failed runs are higher than the sums for the successful runs. This leads to a threshold between failed and successful runs. The possible causes of errors are now combinations of anomalies whose summed magnitude is greater than this threshold.

We evaluate our approach on a simulated quadcopter, which flies a certain route autonomously. During this flight three different types of anomalies can occur: a gust of wind, wrong GPS inputs or insufficient speed. The intensity of the anomalies is determined randomly. The simulation is performed using ArduPilot, a software suite for developing and simulating autonomous unmanned vehicle systems [5]. For the evaluation we compare different classifiers and regressors to determine the best setting for each step. We show that for each step there is an appropriate model that delivers convincing results, assuming that the previous steps deliver perfect results. For step 1, for example, we found a model which reached an Matthews correlation coefficient (MCC) value of about 0.84 with a rather small training set consisting of about 61000 flights. In addition, the evaluation of the successive execution of all steps shows that the approach is functional, albeit in this case the previous steps deliver no perfect results. Using the result of step 1, for example, the approach achieves a subset accuracy of more than 80% at step 2, which is quite decent, considering that it is a rather strict metric for multi-label classification problems.

Chapter 2

Background

2.1 ArduPilot

Ardupilot is an open source software suite for the creation of autonomous unmanned vehicle systems, including drones, helicopters, rovers, boats and more [5]. It was initially developed by Jordi Munoz and Chris Anderson and got its first release in 2009 with version 1.0, which was mainly intended for do-it-yourself (DIY) drones [6]. Within the following years the project has grown steadily and added support for more and more vehicle types. However, ArduPilot was written only for arduino based microcontrollers. Between 2013 and 2014, efforts were made to make it compatible with a range of hardware platforms, such as linux based flight controllers [7]. Today, ArduPilot is used by many different companies and amateurs around the world [8].

2.1.1 Software in the Loop (SITL)

To test an embedded system in a non-critical environment, the hardware in the loop (HITL) and software in the loop (SITL) simulations have proven to be useful. In a HITL simulation the electronic controller is connected to a HITL-Simulator which simulates the environment of the controller [9]. In contrast to that, in SITL the controller is run on the same hardware as the simulator. ArduPilot provides a software in the loop simulator [10], which allows us to run an unmanned vehicle within a simulated environment without any additional hardware. This is possible, since ArduPilot is compatible with a variety of hardware platforms. The sensor data is provided by a so called flight dynamics model (FDM) in a flight simulator. An FDM models flight dynamics, which is concerned with predicting and measuring aerodynamic forces on an aircraft [11]. Ardupilot supports simulators for various vehicles, such as multi-rotor aircrafts, ground and underwater vehicles [10].

2.1.2 MAVLink

The Micro air vehicle communication protocol (MAVLink) is a protocol for the communication with drones [12]. It was first released by Lorenz Meier in 2009. It is a lightweight protocol where a sent packet has an overhead of only 14 bytes. Moreover, it supports several microcontrollers, like ARM7 or STM32. In our setting it is used for the communication between the simulated drone and the training data generation environment.

2.1.3 MAVProxy

MAVProxy is a ground control station (GCS) for unmanned aircraft vehicles (UAV) [13]. A GCS allows human control of UAVs and consists of a processing unit, a graphical user interface, a telemetry module to get information from the UAV and a datalink subsystem to communicate with the UAV [14]. MAVProxy is compatible with any UAV that supports the MAVLink protocol.

2.1.4 Dronekit

The Dronekit-Python API allows to write applications in Python which communicate with a vehicle using the MAVLink protocol [15]. While direct communication via the MAVLink protocol is rather low-level, dronekit offers an object-oriented approach. A vehicle object provides information about its telemetry, state and parameter. In addition, it enables direct control of the vehicle or the uploading and execution of missions.

2.2 Machine Learning

2.2.1 Terminology

In the field of machine learning there are many new terms which we introduce in the following by means of an example. Suppose we want a function to return the number that is displayed on a given grayscale image. Each image has a size of 30×30 pixels. Thus the input of the function is a vector in $\mathbb{R}^{900}$. The individual grayscale values in the vector are called *features* and the vector is called the *feature vector* [16]. An invocation of our desired function results in a value between 0 and 9. That is, there are 10 possible results, called *classes*. Therefore the function is called a *classifier* as it *classifies* the input [16]. More formally, a *classifier* is a function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is the input space and $\mathcal{Y}$

is a discrete and finite output space. If $\mathcal{Y} = \mathbb{R}$ then we speak of *regression* [16]. For the example above, we are in search of a *classifier* of type $\mathbb{R}^{900} \to \{0, ..., 9\}$. The goal is to *learn* from data to be able to create an accurate classifier.

Supervised learning is a type of learning where the classifier is determined by example input-output-pairs [16]. Given the set of all possible classifiers $\mathcal{F} = \{f \mid f : \mathcal{X} \to \mathcal{Y}\}$ and the set of all possible training data $T = \{(X, Y) \mid X \in \mathcal{X} \wedge Y \in \mathcal{Y}\}$ the supervised learning algorithm is a mapping $A : 2^T \to \mathcal{F}$. In other words: the learning algorithm receives a set of training data and generates a classifier with the help of it. In our example above, the learning algorithm would get pairs of grayscale images and the displayed number and return a classifier. There are different approaches to learning. The methods we use in our approach are presented in Section 2.2.2. The resulting classifier is evaluated on values not part of the training data, which is called *testing* [16].

Binary vs. Multi-Class Classification

Our example problem described above is a *multi-class classification problem* (or polychotomy), since $|\mathcal{Y}| > 2$. If $|\mathcal{Y}| = 2$ then we speak of a *binary classification problem* (or dichotomies) [16]. Many approaches allow to solve multi-class classification problems directly (see Section 2.2.2). However, it is also a valid strategy to split such problems into several binary classification problems. Popular strategies are called *One-vs-rest* and *One-vs-one* [17].

In the *One-vs-rest* strategy, we create a binary classifier for each class. Let k be the number of classes. Then the approach results in a total number of k classifiers. The classifier returns true if the input matches the class and false otherwise. In addition, it is required that each classifier returns a confidence score. This is necessary if several classifiers report positive results at once. The final class is the label of the classifier that produced the highest confidence score.

In the *One-vs-one* strategy, we create a classifier for all pairs of classes. This means that every classifier can vote for one of the two classes. The final class is determined by a simple majority vote. The number of classifiers is $\frac{k(k-1)}{2}$.

Multi-Label Classification

In *multi-label classification*, a classifier returns not only one but several classes [18]. Classes are called *labels* in this case. Thus, several labels are assigned to an instance. Given $\mathcal{X}$ and $\mathcal{Y}$ as above, a multi-label classifier is a function $f : \mathcal{X} \to 2^{\mathcal{Y}}$. One way to solve multi-label classification problems is to split them into several binary classification problems, one for each label [19]. Each binary classifier decides if the label is assigned

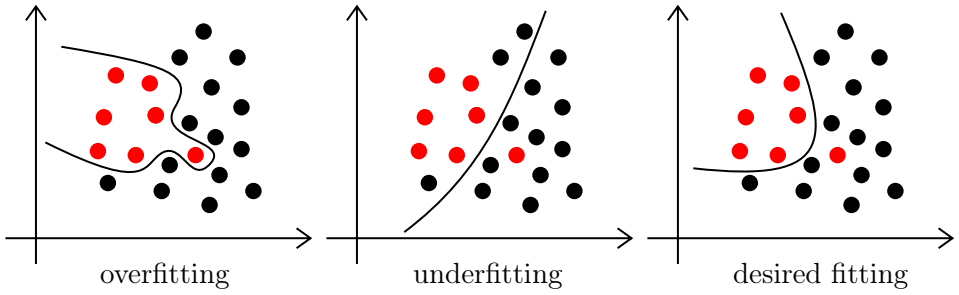

FIGURE 2.1: Overfitting and Underfitting on an example data set with two classes red and black

to the instance or not. This approach is known as the *binary relevance method* (BR). Another possibility is to consider every single element of the power set as a class and to solve the resulting multiclass classification problem [19]. This approach is called the *label powerset method* (LP). This method has the clear disadvantage that the number of classes grows exponentially.

Overfitting and Underfitting

Overfitting and underfitting is a serious problem in machine learning. Usually, data is not exactly separable. There is almost always a certain noise. Overfitting takes this noise into account which causes new data to be classified less accurately as one can see in Figure 2.1. On the left side there is an example where the red dot is still included, although it is in the "black zone". Underfitting on the other side contains unnecessarily many black dots in the "red zone" as one can see in the middle example. The desired partitioning is provided on the right side. It ignores the outlier and otherwise separates the data accurately.

2.2.2 Classifiers/Regressors

This section describes in detail the classification and regression approaches we use in this paper.

k-Nearest-Neighbor

In *k-Nearest-Neighbor* (kNN) the k nearest training points determine the class of the input [20]. It is assigned to the class that occurs most often among the k nearest neighbors. Figure 2.2 shows an example for k-Nearest-Neighbor with two features only. There exist four classes red, yellow, green and blue. Now assume that $k = 8$ and that the pink point

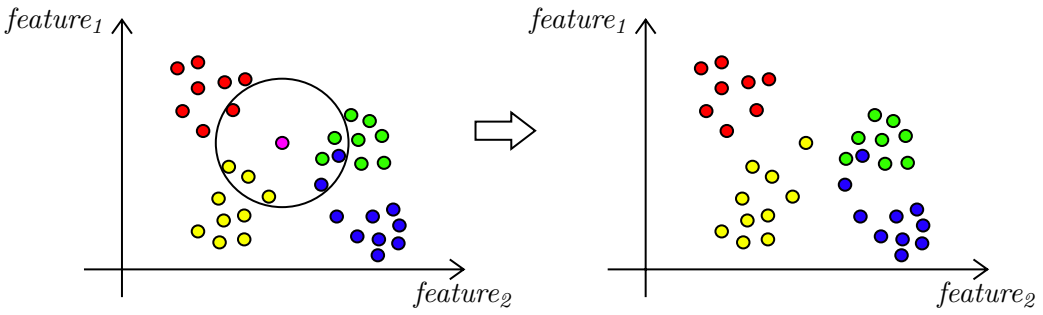

FIGURE 2.2: Example k-Nearest-Neighbor with $k = 8$, two features and four classes red, green, yellow and blue. The pink point is assigned to the yellow class since it occurs most often among the 8 nearest neighbors

shall be assigned to one of the four classes. The eight nearest neighbors are encircled in the figure. The most common class is the yellow one, which is why the pink dot is finally assigned to this class. This example already shows that k-Nearest-Neighbor naturally supports multi-class classification and no adjustments are necessary.

k-Nearest-Neighbor regression is done almost in the same way. But instead of choosing the most frequently occurring class, we compute the mean of the k nearest neighbors [21]. There exist different metrics for the computation of the distance between two vectors [20]. One example is the *Euclidean distance* which is computed as follows. Let $a = (a_1, a_2, .., a_n)$ and $b = (b_1, b_2, ..., b_n)$. Then the Euclidean distance is defined as:

$$d(a, b) = \sqrt{\sum_{i=1}^{n} (b_i - a_i)^2}$$

Another metric is the *Manhattan distance*, which describes the absolute distance of the individual features:

$$d(a, b) = \sum_{i=1}^{n} |b_i - a_i|$$

The generalization of both metrics is called the *Minkowski distance* which is defined as follows:

$$d_\lambda(a, b) = (\sum_{i=1}^{n} |b_i - a_i|^\lambda)^{\frac{1}{\lambda}}$$

For $\lambda = 1$ it equals the Manhattan distance and for $\lambda = 2$ it equals the Euclidean distance.

We use the k-Nearest-Neighbor as a starting point because it is very intuitive, easy to implement but still very powerful. With the k-Nearest-Neighbor analysis, the learning phase is completely skipped. All work is done during classification. This is also the drawback of k-Nearest-Neighbor. The classifier has to compute the distance from the input point to all training points, which is very expensive.

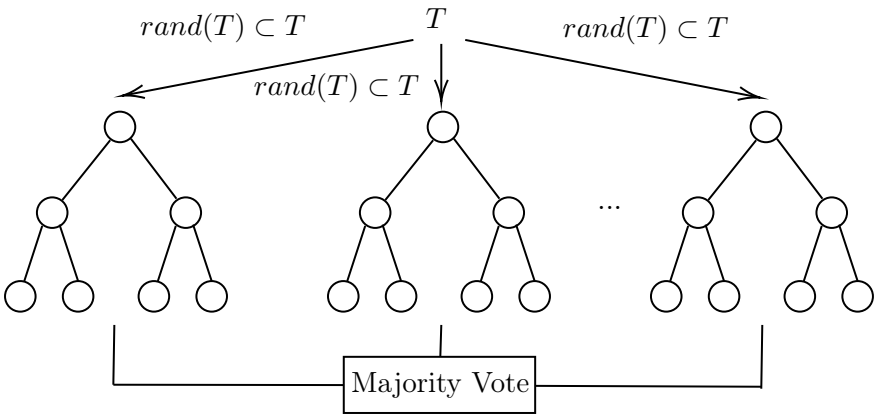

FIGURE 2.3: Graphical representation of the functionality of random forests. Each tree is created with a random subset with replacement of a training set T. During classification each tree decides for one class. The final class is determined by a majority vote.

Random Forest

In *random forests* [22] the learning algorithm creates multiple *decision trees*. Each tree is created with a random sample with replacement out of the training set. Figure 2.3 exemplarily shows the procedure for random forests. Given a training set T we create random subsets of this set and use them to create a decision tree. In the final classifier each decision tree decides for one class. The final class is determined by a simple majority vote. In case of a multi-class classification problem the same procedure is applied: The class with the highest number of votes is the final class.

There exist multiple algorithms for the creation of a decision tree. CART (Classification and Regression Trees) [23], for example, constructs binary trees by choosing a feature and a threshold such that the *Information gain* (IG) or the *Gini gain* is maximized [24]. In more detail, for each node a random sample of features is selected which are possible candidates for the split criterion. One of these features is chosen for a binary split at a determined threshold. There are several approaches to evaluating the quality of a split. We will present the aforementioned Information gain and Gini gain as described in [24, 25]. First of all, let T be the training set with elements of the form $(x, y) = (x_1, x_2, ..., x_n, y)$ and let t be the threshold for the split. Furthermore, let $S_{\leq t}(a) = \{(x, y) \in T \mid x_a \leq t\}$ be the set of all training data, where $x_a \leq t$ and $S_{>t}(a) = \{(x, y) \in T \mid x_a > t\}$ be the set of all training data, where $x_a > t$.

Let C be the set of classes. Then the entropy is defined as $E = -\sum_{i \in C} p_i log(p_i)$, where p_i is the probability that a random choice in T results in an element of class i. Formally the Information gain is defined as follows:

$$IG(T, a) = E(T) - E(T|a)$$

where the conditional entropy is defined as:

$$E(T|a) = \frac{|S_{\leq t}(a)|}{|T|} \cdot E(S_{\leq t}(a)) + \frac{|S_{>t}(a)|}{|T|} \cdot E(S_{>t}(a))$$

In other words, the information gain is defined as the difference between the entropy of the current training set T and the conditional entropy of T given the feature chosen for the split. The higher the Information gain, the more Information we get from this feature. The chosen feature is the one with the highest Information gain.
The Gini gain is computed similarly. In fact, all we have to do is replace entropy with Gini impurity. The Gini impurity is computed as follows:

$$G = \sum_{i \in C} p_i \cdot (1 - p_i)$$

where p_i is again the probability that a random choice in T results in an element of class i. Then the Gini gain is computed similarly

$$GG(T, a) = G(T) - G(T|a)$$

where the conditional Gini is defined as:

$$G(T|a) = \frac{|S_{\leq t}(a)|}{|T|} \cdot G(S_{\leq t}(a)) + \frac{|S_{>t}(a)|}{|T|} \cdot G(S_{>t}(a))$$

Random forests can be used for regression, too. But instead of Gini impurity or entropy we make use of, for example, the mean squared error [25, 26]. Given the current training set T of the current node it is computed as follows:

$$\overline{y} = \frac{1}{|T|} \sum_{(X,y) \in T} y$$

$$MSE = \frac{1}{|T|} \sum_{(X,y) \in T} (y - \overline{y})^2$$

For a leaf node $\overline{y}$ is the result value of the tree. The final result is the average of the results of all decision trees.
Random forests solve the overfitting problem of single decision trees. Since each decision tree has only a subset of all information, they return a more inaccurate result. However, together with the results of all other trees this represents an accurate solution.

Artificial Neural Networks

According to Samarasinghe [27] an artificial neural network is a network of nodes, so called *neurons*. A neuron has arbitrary many input edges and one output edge. There are different types of neurons. *Perceptrons*, for example, output either 0 or 1, whereas *Sigmoid* neurons return any value between 0 or 1. Each input gets a *weight* which models, intuitively speaking, the importance of this input to the output. Furthermore each neuron has a *bias*. The greater this value the closer the result gets to the value 1 or, in the case of perceptrons, the higher the probability to output a 1. Figure 2.4 shows an example neuron with three input edges and one output edge. In case of Sigmoid neurons the computation of y is done in the following way:

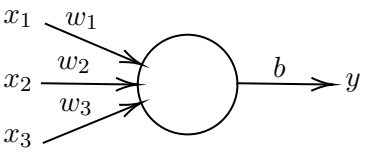

FIGURE 2.4: Example Neuron with three inputs x_1, x_2, x_3. w_i are the weights, b the bias and y is the output

$$y = \frac{1}{1 + exp(-\sum_j w_j x_j - b)}$$

A *Multi-layer Perceptron network* consists of an *input layer*, an *output layer* and a *hidden layer*. The input layer provides the input to the network, wheras the output layer is the output of the network. The number of neurons in the hidden layer can be adjusted as desired. In fact, several hidden layers can be used. Such networks are called deep neural networks [28]. However, Cybenko [29] showed that networks consisting of a finite number of sigmoid neurons and only one hidden layer can approximate continuos functions. Therefore, and to keep the search for a suitable network simple, we will only concentrate on networks with a single hidden layer. Figure 2.5 shows an example network. This network is also an example of a network that can solve multi-class classification problems, because the output layer consists of more than one neuron. For example, each neuron represents a class and the neuron with the highest value for an arbitrary input represents the final class. The goal of the learning algorithm is to adjust the weights and biases such that the *classification error* with the training set is minimized. To achieve this, we need a way to quantify the classification error. It is indicated by a so-called *loss function*. A loss function L is a function of type $\mathcal{Y} \times \mathcal{Y} \rightarrow [0, \infty]$. It evaluates the performance of a classifier. In the following we will define the log-loss function (or cross-entropy) [17]. Let T be the training set, C the set of classes, p_{ij} the probability that sample i is classified with class j, y_{ij} a binary value that is 1 if sample i actually has class j and 0 otherwise.

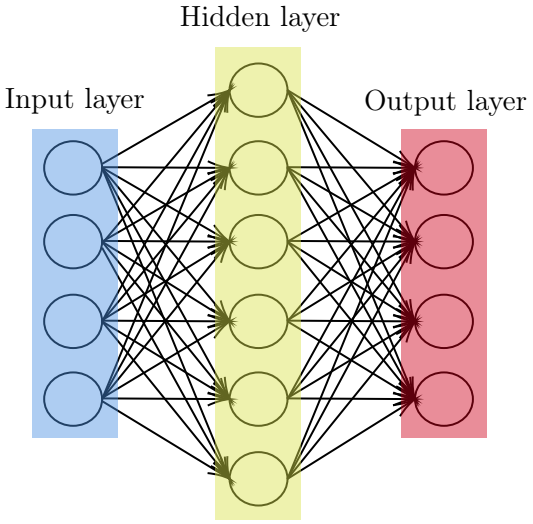

FIGURE 2.5: Example of a Neural Network with Input layer in blue, output layer in red and one hidden layer in yellow.

Then:

$$logloss = -\frac{1}{|T|}\sum_{i=1}^{|T|}\sum_{j=1}^{|C|} y_{ij} log(p_{ij})$$

is the log-loss function. To avoid overfitting one makes use of the regularization method. This technique adds a so-called regularizer to a loss function. In this case, we want to penalize high values for the weights. This is e.g. achieved with the so-called L2 Regularization [30] which is defined as follows. Let $\alpha \in \mathbb{R}_{\geq 0}$ and $w_1, ..., w_n$ be the weights of the network. Then:

$$L2 = \alpha \sum_{i=1}^{n} w_i^2$$

The final loss is therefore $logloss + L2$. The choice of α is problem-specific. One gets suitable values by experimenting.

Now that we are able to quantify the performance of the classifier, we still need a way to improve it step by step. To improve means to minimize the loss, which means that we have to determine the minimum of the loss function. *Gradient descent* [27, 31], for example, is an approach that uses the negative gradient to find local minima of a function f beginning at a starting point x_0. The subsequent values can then be calculated as follows:

$$x_{n+1} = x_n - \gamma_n \nabla f(x_n)$$

where γ_n is the step size. It can be redefined in each step and should neither be too long nor too short. Figure 2.6 shows an example. Here, the step size decreases the closer we get to the minimum in order to avoid overshooting. Figure 2.7 shows an example with large step sizes. One can see that we do not reach the minimum as we are jumping back

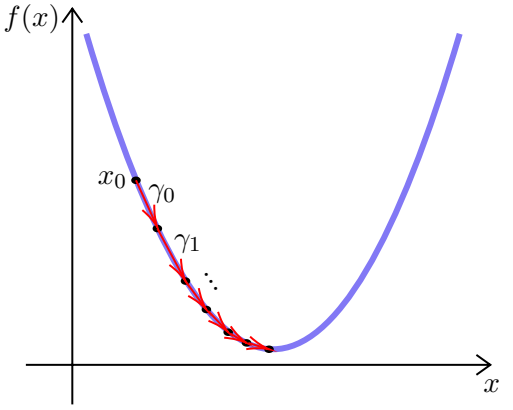

FIGURE 2.6: Example Gradient Descent with suitable step sizes γ_i and starting point x_0

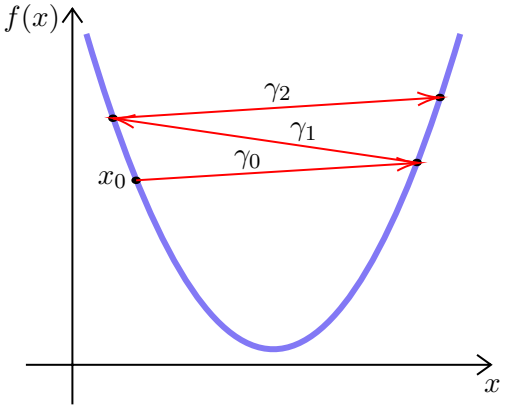

FIGURE 2.7: Example Gradient Descent with large step sizes γ_i and starting point x_0

and forth over it each time. A procedure that has proven to be useful is the so-called momentum [32]. Let $\eta \in [0, 1]$ be the momentum coefficient. Then the updates are now computed as follows:

$$v_{t+1} = \eta v_t - \gamma_n \nabla f(x_n))$$

$$x_{n+1} = x_n + v_{t+1}$$

Intuitively speaking, it moves us faster to the minimum of the function as the step sizes increase. For $\eta = 0$ this is the same as gradient descent. The choice of the coefficient is again a matter of trial and error. *Stochastic gradient descent* [31] is a very similar approach with the difference that instead of considering the entire training set, we consider only a random subset of the training data for the computation of the loss in each new step. This results in more steps being needed, but the calculation of the individual steps is much faster. The calculation of the gradient is usually achieved with the help of a *backpropagation algorithm*. The interested reader is referred to Samarasinghe [27] for more information on this subject, as the details are not relevant for this work.
Another approach is *Newton's Method* as described by Kelley et al. [33]. Usually Newton's method is an approach used to approximate the roots of a function. Starting from x_0, the subsequent values can be calculated as follows until an adequate accuracy has been achieved:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - f(x_n)[f'(x_n)]^{-1}$$

The procedure can also be used to calculate the position of an extremum by searching for the zeros of the first derivative:

$$x_{n+1} = x_n - f'(x_n)[f''(x_n)]^{-1}$$

In the multidimensional case we have to replace the first derivative by the gradient and the second derivative by the Hessian matrix:

$$x_{n+1} = x_n - \nabla f(x_n)[H(f(x_n))]^{-1}$$

One disadvantage of Newton's method is that the calculation of the Hessian matrix is very expensive. Therefore there are approaches like the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [34–37] which approximate this matrix. Approaches like this are called *quasi-Newton methods*.
Regression is done in a similar way. But instead of the logarithmic loss, we compute the mean squared error as described in the previous section. Furthermore, in regression we have only one neuron in the output layer.
Neural networks are very abstract and it is hard to understand decisions. Therefore it is usually not a good idea to start with a neural network. However, in the more recent past it was shown that neural networks perform very well in many applications. Therefore it seems to be a good idea to make use of them.

2.2.3 Imbalanced Data

Classifiers usually only perform correctly if there are about the same number of samples of each class. If there are only very few elements in a class, there is a risk that these will not be considered. With neural networks and random forests such a circumstance becomes particularly noticeable. Imagine, for example, a scenario in which a binary classifier should recognize whether a person is male or female based on characteristics such as size or hair length. The training data consists of 980 male and 20 female examples. The minimization of the logloss function would now be clearly in favor of the male examples, since the female examples hardly carry any weight. Therefore, the trained classifier would probably not recognize female examples, since according to the training data it has proven to be effective in classifying as male. Different resampling methods have been proposed in the past to mitigate the problem. The most obvious approaches are random undersampling and oversampling [38]. Random undersampling reduces the overrepresented classes by randomly removing items of this class. Random oversampling on the other hand copies elements of the underrepresented class. Another

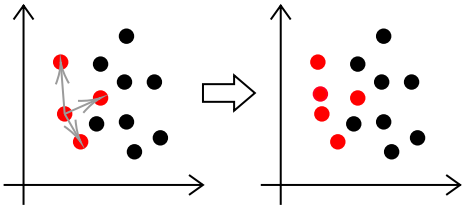

FIGURE 2.8: Example of a SMOTE insertion in the two-dimensional case and $k = 3$ for a underrepresented class red. The upper red dot is randomly selected and inbetween a new red dot is inserted.

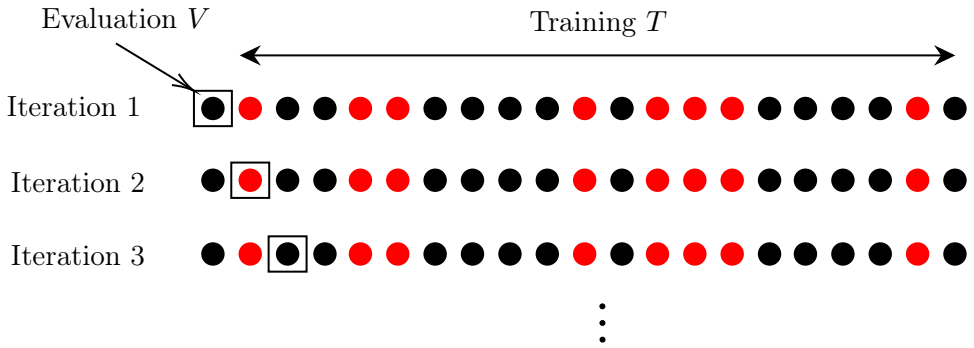

FIGURE 2.9: Example for leave-one-out cross-validation. Only one element is part of the evaluation set in each iteration, the rest is used for training

approach, which seems very suitable for our case, is the synthetic minorityoversampling technique (SMOTE) [39]. This technique determines the k nearest neighbors of an element of the underrepresented class. One of these neighbors is randomly selected. Now a new element is randomly inserted between these neighbors. Figure 2.8 shows an example in the one-dimensional case for $k = 3$. The upper red dot is selected and the new element is placed between them.

2.2.4 Evaluation

Cross-Validation

Cross-validation is a very common technique used to determine more precisely the predictive performance of a model [40]. The original idea was to split a set of samples $S = \{S_1, S_2, ..., S_n\}$ into a training set $T \subset S$ and an evaluation set [41] $V = S \setminus T$. Later on a whole series of splitting techniques was introduced.
The most common technique is the so-called leave-one-out cross-validation (LOOCV) [41, 42]. With this technique n passes are generated. For each pass i a training set $T = \{S_1, ..., S_{i-1}, S_{i+1}, ..., S_n\}$ and an evaluation set $V = \{S_i\}$ is created. Figure 2.9 shows an example. In practice, however, this approach is too expensive. A technique more frequently used in practice is the k-fold cross-validation [41]. With this approach, the data is divided into k equal sized sets. In each iteration one of these sets is chosen for evaluation and the rest for training. Figure 2.10 shows an example for $k = 3$. *Stratified* k-fold cross-validation is a variation of k-fold cross-validation in which the data is split in such a way that the probability of obtaining an element of class c is approximately equal in every set [43]. The example in Figure 2.10 fulfills this property: each set contains about the same number of red and black dots.

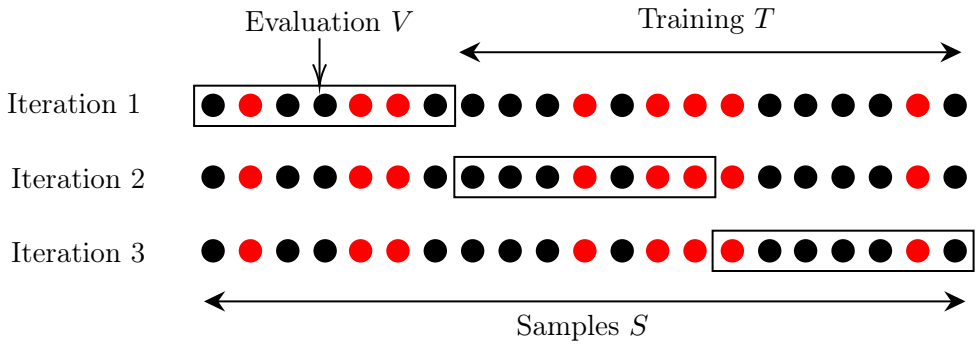

FIGURE 2.10: Example of a k-fold cross-validation with $k = 3$.

Hyperparameter optimization

Determining the correct parameters for each classifier is quite difficult. In fact, it is often recommended to guess several suitable parameters and test the performance with them. For example, to estimate the performance of k-nearest neighbor we check the performance for different k, e.g. $\{5, 8, 11, 14\}$. This technique is called *grid search* [43]. The performance is usually checked with the help of cross-validation.

Classification

There are many approaches to the evaluation of a classifier. The methods listed below and many more have been assembled in detail by Sokolova et al. [44]. We start with the evaluation of a binary classifier, because the evaluations of multi-label classifiers are based on this. First of all we need four basic building blocks, as depicted in Table 2.1.

TABLE 2.1: A confusion matrix

Correct Class \ Predicted Class	Positive	Negative
Positive	true positive (tp)	false negative (fn)
Negative	false positive (fp)	true negative (tn)

Inputs that have been correctly classified as positive by a binary classifier are called true positives (tp). True negatives (tn) are those that have been correctly classified as negative. On the other hand, false positives (fp) are those that have been wrongly classified as positives and false negatives (fn) have been wrongly classified as negative. In a first step we can compute the accuracy, which is the percentage of predictions that are correct:

$$\frac{tp + tn}{tp + tn + fp + fn}$$

Furthermore we can define the positive predictive value (PPV or precision) and the true positive rate (TPR or recall) as:

$$precision = \frac{tp}{tp + fp} \ , \quad recall = \frac{tp}{tp + fn}$$

Precision indicates how many of the items classified as positive are actually positive. Recall indicates how many positives have been classified as positive. Analogously we can define the negative predictive value (NPV) and the true negative rate (TNR or specificity) as:

$$NPV = \frac{tn}{tn + fn} \ , \quad TNR = \frac{tn}{fp + tn}$$

These values already provide a detailed insight into the performance of a binary classifier. However, it would be preferable to evaluate the performance with only one value. Very often the so-called F1 score is used for this purpose, which is the harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{PPV \cdot TPR}{PPV + TPR}$$

Yet this score has to accept criticism because it disregards the true negative rate. A solution is offered by the Matthews correlation coefficient (MCC) [45], which is calculated as follows:

$$MCC = \frac{tp \cdot tn - fp \cdot fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}$$

It returns a value between -1 and 1. If it is 1 then the classifier performs perfect. If it is 0 then it is no better than a random classifier and if it is -1 it performs exactly opposite to the perfect classifier.

With multi-label classification one can proceed similarly. The following descriptions are explained in detail by Zhang et al. [46]. First of all we need to calculate the confusion matrix for each label. Now, either a binary classifier can be applied to each confusion matrix and the average is calculated from these results, which is called macro-averaging. Or we add up the confusion matrices and apply a binary metric to the result, which is known as micro-averaging. More formally, let $L = \{l_1, ..., l_n\}$ be the set of labels. Let $tp_{l_j}, fp_{l_j}, tn_{l_j}, fn_{l_j}$ be the true positives, false positives, true negatives and false negatives of label l_j. Let $binary \in \{precision, recall, NPV, TNR, F_1\}$. Then:

$$binary_{macro} = \frac{1}{|L|} \sum_{l \in L} binary(tp_l, fp_l, tn_l, fn_l)$$

is the macro-average value and:

$$binary_{micro} = binary\left(\sum_{l \in L} tp_l, \sum_{l \in L} fp_l, \sum_{l \in L} tn_l, \sum_{l \in L} fn_l \right)$$

is the micro-average value. Intuitively speaking macro-averaging weights all labels equally and micro-averaging weights all examples equally.
Another rather strict metric is called *subset accuracy*. Let N be the number of evaluation inputs, y_i the correct value of evaluation input i and $\hat{y}_i$ the predicted value of i. Then:

$$\mathbb{1}_\pi = \begin{cases} 1, & \textit{if } \pi \textit{ holds} \\ 0, & \textit{otherwise} \end{cases}$$

$$\textit{subsetacc} = \frac{1}{N} \sum_{i=0}^{N-1} \mathbb{1}_{y_i = \hat{y}_i}$$

In other words, subset accuracy specifies in percent the number of inputs for which all labels are correct which is rather strict compared to the other metrics.

Regression

For regression we need different evaluation models. A metric which is suitable for this and which we already know is the mean squared error. Let N be the number of evaluation inputs, y_i the correct value of evaluation input i and $\hat{y}_i$ the predicted value of i. Then we can compute the mean squared error as follows [16]:

$$MSE = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - \hat{y}_i)^2$$

Analogously we can compute the mean absolute error (MAE) [47]:

$$MAE = \frac{1}{N} \sum_{i=0}^{N-1} |y_i - \hat{y}_i|$$

In some cases it might also be useful to get the maximum error (ME) and the median absolute error (MedAE) [48], which are computed as follows:

$$ME = max(|y_i - \hat{y}_i|) \ , \quad MedAE = median(|y_1 - \hat{y}_1|, ..., |y_1 - \hat{y}_1|)$$

An interesting model is the so called R^2 score (or coefficient of determination) [16]. It is computed as follows:

$$R^2 = 1 - \frac{\sum_{i=0}^{N-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{N-1} (y_i - \overline{y})^2}$$

where $\overline{y} = \frac{1}{N} \sum_{i=0}^{N-1} y_i$. A nice intuition for this is provided in Figure 2.11. In linear regression we are in search of a linear function that comes closest to the input data. In the worst case x provides us with no information about y. In this case it is a good estimate to

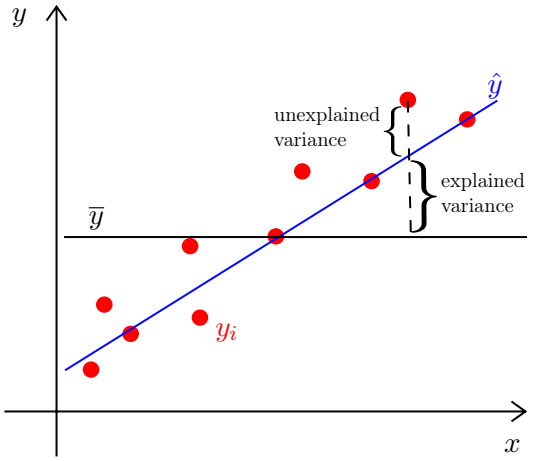

FIGURE 2.11: Illustration of the R^2 score for linear regression. A fraction of the variance can be explained by the slope of the straight line.

determine the average value $\overline{y}$. In our example, however, there is a relationship between x and y: for higher x we get higher y. Therefore it is possible to reduce the variance by adjusting the gradient of the linear function accordingly. The variance that gets lost is called the explained variance, because we were able to "explain" it by the gradient. Yet an unexplained variance remains. R^2 now relates these two values to each other intuitively as follows.

$$R^2 = 1 - \frac{unexplained\ variance}{explained\ variance + unexplained\ variance}$$

Thus a value of 0 means that a classifier is not better than $\overline{y}$ and a value of 1 means that it maps all values perfectly. A value less than 0 means that the classifier classifies even worse than $\overline{y}$.

2.2.5 Scikit-learn

To minimize the development effort, we made use of a programming framework to realize the classification algorithms. For classical machine learning algorithms, like k-Nearest-Neighbor or Random Forests but also for the more advanced methods like neural networks there exists the scikit-learn library for Python [49]. It provides an easy to learn high-level interface and enables the comparison of different methods which makes it an adequate choice.

Based on scikit-learn many more libraries have been published, extending it. To realize multi-label classification we use the scikit-multilearn API [50]. Moreover to tackle the problem of imbalanced data we make use of the imbalanced-learn library [51].

Chapter 3

Related Work

3.1 Multi-label Classification

Multi-label Classification is considered a very difficult machine-learning problem. Therefore a lot of research has been done to transform problems of this kind into simpler problems. These approaches are interesting for our problem as well. However, in the end we decided against them, because our problem can be broken down into even simpler ones. In the following we present a selection of approaches for the decomposition of multi-label classification problems.

Read et al. [19] present classifier chains for Multi-label Classification. The binary relevance method, introduced in Section 2.2.1, has a drawback. If the classes are correlated with each other, this information will get lost during this procedure. Therefore, they propose to create chains of classifiers. Let C be the set of labels and $T = \{(x_1, S_1), ..., (x_n, S_n)\}$ the training set with $S_i \subseteq C$. They define an ordering on the labels and represent S_i by a binary vector $(l_1, ..., l_{|C|}) \in \{0, 1\}^{|C|}$ which indicates whether label l_j belongs to x_i or not. Then they create $|C|$ classifiers $c_1, ..., c_{|C|}$, where the training set for c_i is modified such that $T' = \{(x, l_1, ..., l_{i-1}, l_i) \mid (x, S) \in T\}$. In other words, the classifier c_i gets information about the classifications of the previous classifiers by adding them to the features in the training set. They call this the classifier chain method (CC). The authors themselves criticize that the arrangement of the labels is crucial. Thus, they present another approach which they call ensemble of classifier chains (ECC). ECC trains m CC classifiers. Each CC classifier is trained with a randomly ordered C and a random subset with replacement of T. For the determination of the resulting class they define a threshold t. A label l_i is assigned to an input if the number of classifiers that assign label l_i is greater than t.

Tsoumakas et al. [52] present the RAKEL (RAndomk-LabELsets) algorithm. It is an ensemble method of the label powerset method described in Section 2.2.1. The procedure is as follows. Let $C^k = \{Y \mid Y \subseteq C \wedge k = |Y|\}$ be the set of all subsets of C of size k. Then

RAKEL trains m classifiers $c_1, ..., c_m$ with $c_i : X \to 2^{Y_i}$, where $Y_i \in C^k \setminus \{Y_1, ..., Y_{i-1}\}$. In other words, for a given k and m it creates m LP classifiers with randomly chosen $Y_i \in C^k$ without replacement. The final labels are determined in the same way as in ECC. For appropriate k and m it was shown that RAKEL performs better than BR and LP.

Fürnkranz et al. [53] make use of label ranking for Multi-label classification. Given a set of labels $C = \{l_1, ..., l_n\}$ the task is to find a ranking $\prec_x$ over C for a given input $x \in \mathcal{X}$ and some input space $\mathcal{X}$. Ranking by pairwise comparison (RPC) achieves this by creating a binary classifier for each pair of labels (l_i, l_j). The authors transform label ranking to Multi-label classification by adding a new label l_0, which serves as a split point between relevant and irrelevant labels. A label l_i is relevant if $l_i \prec_x l_0$ and irrelevant if $l_0 \prec_x l_i$. Relevant labels are labels, that will be assigned to the input and irrelevant labels are those that will not be assigned to the input. This approach provides more information about the relationship of the individual labels to each other. However, this is done at the expense of the runtime which makes it intractable for large problems.

3.2 Runtime Verification

Runtime verification or runtime monitoring is a technique for the verification of a single run of a system. In a sense, this is an extension of testing, since specification languages are able to extract and process information about the state of a running system [4]. One such specification language is called Lola [4] which is stream-based: It gets possibly multiple streams of input data and translates them into output streams. Lola was already used in many different contexts such as synchronous circuits [4], network monitoring [54] and unmanned aircraft systems [55].

According to Adolf et al. [55] a Lola specification is a system of equations over stream variables of the following form:

input T_1 t_1
...
input T_m t_m
output T_{m+1} $s_1 := e_1(t_1, ..., t_m, s_1, ...s_n)$
...
output T_{m+n} $s_n := e_n(t_1, ..., t_m, s_1, ...s_n)$

where $t_1, ..., t_m$ are the input streams, $s_1, ..., s_n$ are the output streams and $T_1, ..., T_{m+n}$ are the types of the respective streams. The stream expressions $e_1, ..., e_n$ have access to input and output streams. The language allows constants, functions, conditionals and

offset expressions for the construction of e_i. An offset expression is an expression of the form $s[-i, d]$, where s is a stream of type T, i is an integer value and d is a value of type T. Let c be the current position in the stream. The offset operator returns the value of the stream at position $c - i$. If this value does not exist, e.g. if $c - i < 0$, it returns the default value d.

The semantics of a Lola specification are as follows. Let $\tau = \langle \tau_1, ..., \tau_m \rangle$ be a valuation of input streams of length $N + 1$. A tuple $\sigma = \langle \sigma_1, ..., \sigma_n \rangle$ of streams of length $N + 1$ describes the set of output streams s_i, if and only if for all $1 \leq i \leq n$ and $0 \leq j \leq N$, $\sigma_i(j) = val(e_i)(j)$ holds. In this case σ is "an evaluation model of the Lola specification for τ" [55]. $val(e)(j)$ is defined as a set of partial evaluation rules, as described for example in [4].

The original design was done by by D'Angelo et al. [4] in 2005 for the runtime monitoring of synchronous systems. The language was extended to be more suitable for network monitoring [54]. The advantage of this extended language, called Lola 2.0, is that incoming data can be subdivided such that smaller amounts of data can be processed. Lola 2.0 is intended to be a bridge between light-weight network monitoring tools, like Snort [56], and more involved approaches based on expensive formalisms such as Bro [57]. Snort is a Network Intrusion Detection System (NIDS) which is able to analyse and log packets based on information given in the payload of the packets. In contrast to that, the specification language of Bro operates on events generated from prefiltered network traffic. The language is as expressive as a programming language. In the more recent past, Lola was integrated into the flight operation framework ARTIS (Autonomous Research Testbed for Intelligent Systems) [55]. It consists of a software framework and a set of unmanned aircraft of different classes and sizes. Faymonville et al. propose RTLola [58], the real time extension to Lola. In RTLola input streams are considered that extend at unknown rates and each new event has a real-valued time stamp. A new offset operator is introduced that aggregates all values over a provided real-time window. For example, $s[2sec, 0, avg]$ computes the average value of all values of the stream that occured during the last two seconds.

Runtime verification is very similar to our approach. We also process the information of a system to extract and evaluate possible anomalies. However, so far our approach can only be applied to the logs after a system run. Runtime verification allows the analysis of logs as well as the analysis during runtime.

3.3 Failure Analysis

Finding the cause of the failure is a hard task since the failure cause usually occurs much earlier. Therefore scientists have always been looking for ways to simplify debugging

tasks. The generic term for such approaches is Failure Analysis. In concrete terms, these are approaches to collecting and analysing data to determine the cause of failure in a system [59]. This is exactly what we are trying to achieve in our approach. In the following we present a selection of approaches that try to simplify or automate Failure Analysis. These approaches are closely related to our approach, as both support engineers during debugging by presenting possible failure causes.

An old technique used until today is the so called "Fault Tree Analysis" [1]. The goal is to find the source of a system failure. This is achieved by creating a graphical representation of causal relationships in the system. Thereby the plan of the system serves as input and events that cause a system failure or malfunction are determined. Then, the task is to find events that lead to the failure. Events can be combined with logic gates, resulting in a circuit. Fault trees are usually created manually and therefore require a lot of time and effort. We also have to determine possible causes of failures in our approach and train the learning algorithms on them. However, our algorithm then tries to determine the cause of the failure on its own, while a fault tree should be regarded as a help for manually finding and preventing causes.

Another subarea of Failure Analysis is fault localization. More specifically, the aim is to find the statement or the set of statements that caused a failure within a program. There exist many different approaches to automated fault-localization of which we will introduce some representatives in the following paragraphs.

In slice-based approaches[60, 61] coverage information of test suites is used. For example, if two tests produce the same error, the faulty statements are most likely the ones used in both tests.

In spectrum-based approaches, the program is divided into multiple entities [62]. These entities are ranked according to a specific formula or statistical information. If many tests that utilize a particular entity fail, the suspiciousness score for that entity increases. A famous representative of these approaches is Tarantula [63]. It provides a visual representation of the source code and colors the lines green to red depending on how likely it is that this line is the cause of the error.

Xuan et al. [64] argue that there exists no optimal ranking metric for spectrum-based fault localization. They make use of machine learning techniques to build a ranking function. The learning algorithm is trained with pairs of erroneous and error-free source code. Their experiments show that their approach performs better than, for example, Tarantula in terms of localizing faults.

Zeller et al. [65] propose a program state based approach for fault-localization. The idea is to spot the differences in states of one passing run and one failing run. Their algorithm reduces these differences to finally obtain a minimal set of possible failure causes [62].

Fault localization approaches are well suited for Failure Analysis of systems working with software suites like ardupilot. However, the system can behave correctly and still fail due

to external influences. Such failure causes are difficult to detect with these approaches. An exception to this is the approach of Zeller et al. By comparing successful and failed runs, the causal analysis could be extended to external causes as well. The step-by-step removal of individual external influences should allow the actual causes to be found. However, this means that the complex system must be run under the new conditions again and again in order to analyze the behavior of the system without certain anomalies. This is difficult to achieve with systems such as unmanned aircraft vehicles.

3.4 Program Intrusion Detection

Intrusion Detection approaches try to detect anomalous program behaviour, which can be done e.g. based on simple pattern matching or with the help of machine learning. The machine learning step of our approach is very similar to machine learning based program intrusion detection. Both try to find anomalous behaviour in a trace. However, in contrast to our approach, this is only a binary classification problem: One simply wants to find out if there is an intrusion or not. Nevertheless, our approach is strongly inspired by program intrusion detection. In the following we present intrusion detection algorithms based on machine learning.

Ghosh et al. [66] present three different intrusion detection algorithms. The first one is based on simple pattern matching. The second one makes use of a simple feed forward multi-layer perceptron network and the last one uses more complex Elman networks. The authors argued that it is sufficient to capture system calls of a program as a compact representation of its behaviour. They are captured by using Sun Microsystem's Basic Security Module (BSM) auditing facility for Solaris[1].

The feed-forward network gets as input a sequence of system calls and decides whether this sequence is anomalous or normal. To classify a session, which is an execution of multiple programs, they make use of the leaky bucket algorithm, which accumulates outputs of the neural network but also slowly leaks its value. Thus, if anomalous behaviour becomes more frequent in a short time it will raise a flag.

The problem with the described algorithm is that it is not able to capture intrusions spread over a longer period of time, since it is only able to decide for small parts of a sequence whether it is anomalous or not. Thus, the authors proposed an Elman network, which has so called context nodes in the hidden layer. A context node gets input from one node in the hidden layer and returns its output to each node in the layer of the context node. This Elman network now predicts the next sequence of events which is compared to the actual sequence of events. The difference of these sequences is the measure of anomaly. Finally, the leaky bucket algorithm is used for the results as described above.

[1] see e.g. https://docs.oracle.com/cd/E19455-01/806-1789/806-1789.pdf

The Elman network performed best, while the feed forward network was not much better than a simple pattern matching approach.

3.5 Causality Analysis

The causality analysis of systems is an ongoing research topic, where many different approaches have been proposed. These approaches are very similar to our approach, as the goal is always to simplify the process of finding the causes of faulty behavior in complex systems. In the following we will present some of them.

The method of Wang et al. [67] uses a hybrid approach to causal analysis to find the fault-causing components in a safety-critical system. They choose a counterfactual approach by algorithmically deriving alternative system behavior, however without re-executing the system. The underlying idea is to consider how the system would have behaved if a component had not exhibited malfunction. The authors show that their approach is more accurate if they emulate the execution of individual components and if they additionally consider input and output relationships between the components.

Kuntz et al. [68] developed an approach for the automatic creation of fault trees based on probabilistic counter-examples. The method aims to facilitate the reliability analysis of safety-critical systems. They make use of a probabilistic model checker to produce counterexamples for interesting properties. These counterexamples consist of system execution paths and the associated probability information. The counter-examples serve as a basis to derive fault trees by extracting the causal relationships. None of the analysis steps require manual intervention.

Gössler et al. [69] propose a semantic framework to improve troubleshooting in complex component-based computer systems. Their counterfactual analysis is based on configuration structures with the aim to handle partial and distributed observations and concurrent systems uniformly. In addition, it should identify necessary and sufficient causes of faults within systems.

Chapter 4

Approach

In this chapter we provide a more detailed overview of our approach. First, we describe in detail how the data for the machine learning approaches is created. Then we describe our approaches for the anomaly detection and type identification. After that, we explain our methodology for the causality analysis.

4.1 Creation of Training Data

For the generation of the training data we use the SITL simulator integrated in ArduPilot. This allows us to realize the whole simulation in one place. Furthermore, we are able to run multiple simulations at once, whereas in a HITL simulation we would require multiple electronic controllers to achieve the same result. The autonomous copter is a quadcopter with default parameters as described in Appendix A.

4.1.1 Mission

The drone has to accomplish a mission which is provided by a so called waypoint (WP) file as one can see in Figure 4.1. The general format is as follows [70]:

```
1 QGC WPL <VERSION>
2 <INDEX> <CURRENT WP> <COORD FRAME> <COMMAND> <PARAM1> <PARAM2> <PARAM3> <PARAM4>
      <PARAM5/X/LONGITUDE> <PARAM6/Y/LATITUDE> <PARAM7/Z/ALTITUDE> <AUTOCONTINUE>
```

We used the MAVLink commands *NAV_TAKEOFF* (22), *NAV_WAYPOINT* (16) and *NAV_RETURN_TO_LAUNCH* (20). Thus, the drone first takes off to the provided latitude, longitude and altitude. Then it processes the given waypoints one after the other and finally returns to the initial location. Figure 4.2 shows a graphical representation of the mission.

The mission is initialized as follows. A simulated quadcopter is set to the position

```
QGC WPL 110
0 0 3 22 0.000000 0.000000 0.000000 0.000000 49.254986 7.040858 50.000000 1
1 0 3 16 0.000000 0.000000 0.000000 0.000000 49.257015 7.040343 50.000000 1
2 0 3 16 0.000000 0.000000 0.000000 0.000000 49.259010 7.051792 50.000000 1
3 0 3 16 0.000000 0.000000 0.000000 0.000000 49.252594 7.041029 50.000000 1
4 0 3 16 0.000000 0.000000 0.000000 0.000000 49.253616 7.037252 50.000000 1
5 0 0 20 0.000000 0.000000 0.000000 0.000000  0.000000 0.000000  0.000000 1

```

FIGURE 4.1: WP file of the mission that the drone has to accomplish. The first line describes the format and the version. Every following line contains the command to be executed at the fourth position, the target location at fourth and third last position and the altitude at the second last position.

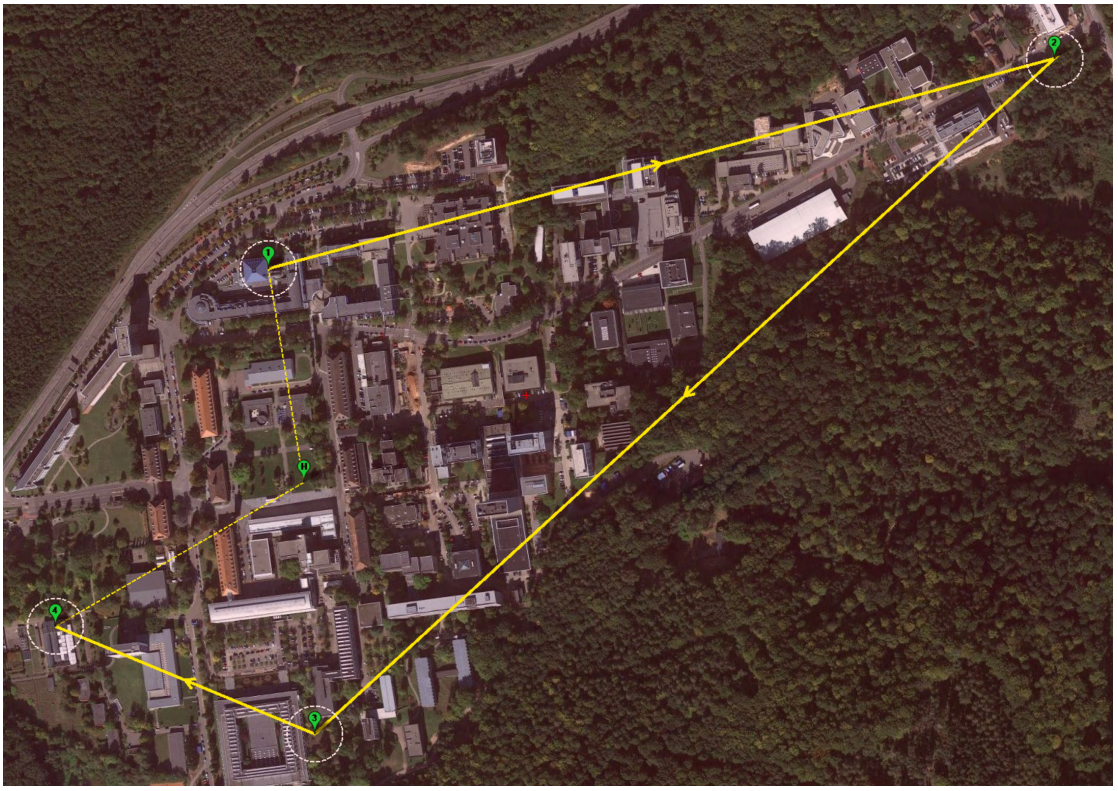

FIGURE 4.2: A graphical representation of the mission from Mission Planner [71]. The drone takes off at position H(ome) and then traverses waypoints 1 to 4 one after the other to finally return to H.

(49.255070, 7.040825) on Saarland University at an altitude of 0 meters. We use dronekit to communicate with the drone via MAVLink. With the help of dronekit we upload the mission to the drone and set the vehicle mode to AUTO such that the drone can accomplish its mission.

4.1.2 Anomalies

Under perfect conditions the drone is able to accomplish the mission without issues. However, since our goal is to find anomalous behaviour, we must manually intersperse

abnormal behavior. Fortunately, the SITL simulator provides support for the simulation of many different environmental influences. We have opted for three different possible anomaly types which we describe in the following.

Speed

The first possible anomaly is that the drone is flying too slowly. In this case it is not possible for the drone to complete its mission before the battery drains. During the flight we randomly set the speed of the drone and wait a random amount of time for the next speed change. We want higher probabilities for higher speeds and time intervals between two speed changes. Intuitively speaking we fold the symmetrical parts of the normal distribution on top of each other and move the result to the right by a given amount. To this end, we define the following function for a random variate x of a normally distributed random variable $X \sim \mathcal{N}(0, \sigma^2)$ and a margin y:

$$r(x, y) = \begin{cases} x + y, & -y < x \leq 0 \\ y - x, & 0 < x < y \\ 0, & x \geq y \ \vee x \leq -y \end{cases}$$

The drone has a maximum speed of $15\frac{m}{s}$. We decided for a normally distributed $X \sim \mathcal{N}(0, 6)$ for the random choice of speed. A new random value for speed is now the result of $r(x, 15)$, where x is a random variate of X. For the waiting time in between we proceed in the same way. For a normally distributed random variable $X \sim \mathcal{N}(0, 200)$, the waiting time between two speed changes is the result of $r(x, 600)$, with x being a random variate of X. Figures 4.3 and 4.4 show the probability distribution for the random choice of the speed and the waiting time in between. The probability density function for the computation of the speed value is $f(x) = 2 \cdot f_{\mathcal{N}(0,6)}(x - 15)$ for $0 < x \leq 15$ and similarly for the waiting time it is $f(x) = 2 \cdot f_{\mathcal{N}(0,200)}(x - 600)$ for $0 < x \leq 600$. As desired, a higher speed gets a higher probability, since in the end we want more successful flights than failed ones. The waiting time in between is with high probability very long, as an autonomously flying drone is usually not configured in such a way that it constantly changes its speed on a straight route.

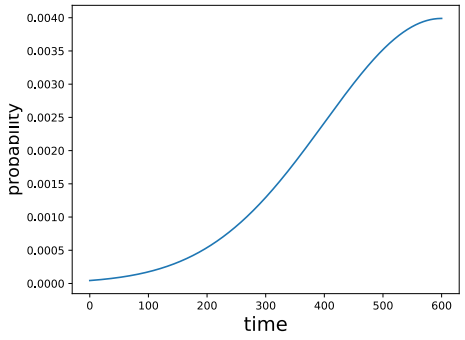

FIGURE 4.3: Probability distribution for the time between two random speed changes in seconds: $f(x) = 2 \cdot f_{\mathcal{N}(0,200)}(x - 600)$

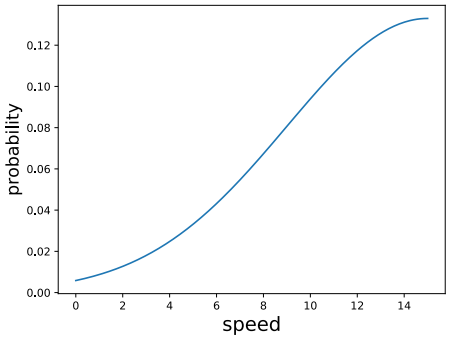

FIGURE 4.4: Probability distribution for the random determination of speed in $\frac{m}{s}$: $f(x) = 2 \cdot f_{\mathcal{N}(0,6)}(x - 15)$

Wind

Wind is another possible anomaly type. We adjust two parameters. First, the wind speed in $\frac{m}{s}$, and second the horizontal direction. We proceed in a similar way for the wind speed as we did for the drone speed. For a normally distributed $X \sim \mathcal{N}(0, 6)$ the new value for the wind speed is the result of $20 - r(x, 20)$, for a random variate x of X. Figure 4.5 shows the probability distribution for the wind speed. It is described by the following probability density function: $f(x) = 2 \cdot f_{\mathcal{N}(0,6)}(x)$ for $0 \leq x < 20$. Similar to the speed anomaly we wanted a higher probability for lower wind speeds. In contrast to that, the wind direction is determined by a discrete uniform distribution, since we wanted to assign the same probability to each direction. After a waiting time, which is determined by a uniform distribution between 0 and 20 seconds, the wind speed is determined again.

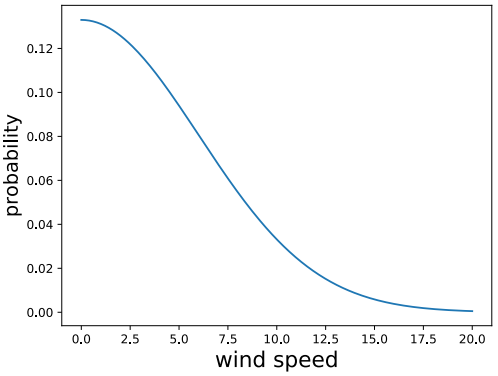

FIGURE 4.5: Probability distribution for the random determination of wind speed in $\frac{m}{s}$: $f(x) = 2 \cdot f_{\mathcal{N}(0,6)}(x)$

GPS Glitches

Last, we decided for GPS glitches as the third type of anomaly. During a GPS glitch the position of the drone is incorrectly determined by the GPS module. In our case during a

GPS glitch, latitude and longitude differ from the correct value by $0.0002 - 0.002$ each, which corresponds to a displacement of about 25 to 265 meters. Any possible displacement may occur with the same probability. A GPS glitch appears with a probability of 0.01. Between two possible GPS glitches a time between 0 and 100 seconds elapses, randomly distributed.

4.1.3 Extraction of Relevant Data

MAVProxy stores any received telemetry from the simulated vehicle to a log file. This log file is converted to a json file with the help of a tool provided by MAVLink. The file provides all information about the flight which we need for our approach. However, for a later classification we need to convert the data into a sequence of numbers. Furthermore, we must filter out the most relevant information from the large amount of information available.

Each message in the telemetry log is timestamped. We want to demonstrate our extraction on an example as shown in Figure 4.6. Suppose we want to extract the heading and the altitude of the drone at a sampling interval of 0.1 seconds. Whenever a new value for a variable like the heading appears at a given timestamp, all variables are aligned up to this point and the new value is inserted. The first values for heading and altitude appear at timestamp 0.0. The first new value for the heading appears at time 0.63. Thus, we need to add six times the value of the current value to both arrays. Then we replace the last value of heading by the new value, which is 46. Altitude is updated at time 0.81. Again we need to add two times the value of the current value to both arrays and replace the last value of altitude by 49. In the following we will refer to such sequences of data as streams. The described technique is known as 0-order hold. Another possibility would have been interpolation. With this technique, however, we would have the problem that the positions on GPS glitches slowly adapt to the incorrect position instead of having a sudden jump which makes the classification more complicated.

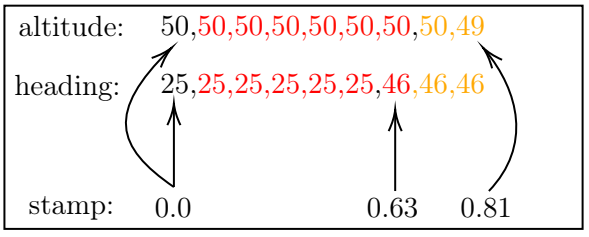

FIGURE 4.6: An example of how the data sequences are created. At timestamp 0.0 two values for heading and altitude occur. At 0.63 a new value for heading appears. Up to that point, the previous values will be used. Analogously for timestamp 0.81

Now, several questions remain unanswered. The first question is whether we should choose the same sampling interval for the different flights or a different one for each

flight. The first option has several drawbacks. If a flight takes too long we miss parts of it, as we need an upper limit for the number of samples. If it is too short, we have to fill it with some data that does not exist. We decided for the second option, which means that we chose a different sampling interval for each flight. Let s be the number of samples we want to have and t the time that it took for the drone to execute the mission. Then the sampling interval i is $\frac{t}{s}$. The advantage of this method is that we have always sampled the entire flight. The disadvantage, however, is that long flights may be sampled insufficiently because of a high sampling interval. Investigation of the training data has shown, however, that outliers in the sampling interval are rare.
The second question that needs to be answered is how many samples we need. If we have too many samples, then the classification will get very slow. We decided for 600 samples per flight. As a flight takes about 6 minutes, this corresponds to a sampling interval of about 0.6 seconds. This proved to be a good compromise between accuracy and efficiency.
We sample the following data to train and evaluate the classifiers: latitude, longitude and speed information account for GPS and speed anomalies. Moreover to successfully detect wind anomalies we consider pitch, roll and heading of the drone. Additionally, we store some more information which does not serve as input to the classifier, but makes it easier to perform the manual classification. To accurately determine wind anomalies we sample wind speed and wind direction from the simulator. For GPS glitches we store the actual glitch in x- and y-direction. Finally, the battery state and altitude of the drone are added.

4.1.4 Manual Classification of Anomalies

Since we do supervised learning we now have to manually classify the generated training data. Up until now we created the features. As already mentioned, we stored some additional information which makes it easy for us to perform the classification. For the three possible anomalies we create three new streams of size 600, where the values can either be 0 or 1. If it is 1, it means that there is an anomaly of the according type, otherwise not.
Detecting a wind anomaly is straightforward. While the wind speed is greater than 2, we set the value of the stream for the wind anomalies to 1, and 0 otherwise.
Detecting a GPS glitch is a bit more involved. ArduPilot features a built-in glitch protection. Given a previous position, speed and direction it can predict the next position with some precision. If the input of the GPS module deviates significantly from the predicted position, this input is discarded first. The drone is assumed to be at the expected position. Figure 4.7 illustrates the glitch protection mechanism. If the new

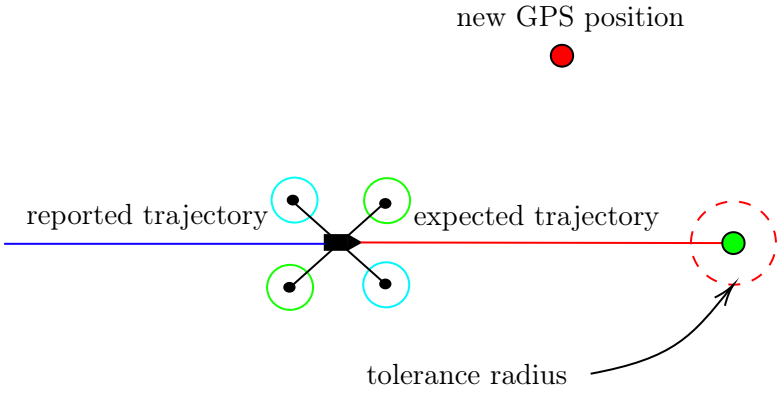

FIGURE 4.7: Ardupilots glitch protection. If the new GPS position is not within the tolerance radius it will be discarded. The tolerance radius increases over time until the new position is again within the radius.

GPS position is within the tolerance radius it is considered. This radius increases over time. The glitch is cleared if the GPS position is within the tolerance radius again. For us, that means it is not enough to check if the GPS glitch values are greater than 0. Depending on how long this glitch lasts, it has not changed the flight behaviour of the drone. Instead we have to check if there is a larger jump in the position. Thus, we check if the distance between two positions is larger than 25 meters, because this is the smallest possible jump during a glitch. Only if this is the case, then there is a GPS glitch at this position.
For the detection of speed anomalies we decided for a reference flight, which ran under ideal conditions. To check if the speed at a certain point is appropriate, we search for the nearest point in the reference flight and check if the speed is at least 90% of the reference speed.

4.1.5 Normalization

Especially for k-nearest neighbor it is reasonable to normalize the data. In this case, for example, a large difference in heading separates the data to a greater extent from each other than a large difference in speed. The maximum speed difference is $15\frac{m}{s}$ and the maximum difference in the heading is 360°. For this reason, neighbors with less difference in heading would be preferred in k-nearest neighbor.
To handle such cases we transform all values such that they are in the $[0, 1]$ interval. The transformation is depicted in Table 4.1. The speed only needs to be divided by the maximum possible speed, which is $15\frac{m}{s}$. However, since the speed can be slightly higher due to gusts of wind we increase this value to 18. To normalize latitude and longitude we first determine the maximum and minimum latitude and longitude from all flights. With the help of these values we can normalize as shown in the table. A

TABLE 4.1: Normalization of different stream values

Values	Variable	Conversion
Speed	s	$\frac{s}{18}$
Latitude	lat	$\frac{lat - min_{lat}}{max_{lat} - min_{lat}}$
Longitude	lon	$\frac{lon - min_{lon}}{max_{lon} - min_{lon}}$
Heading	h	$\frac{h}{360}$
Pitch	p	$\frac{p \mod 360}{360}$
Roll	r	$\frac{r \mod 360}{360}$

nice side effect of this normalization of latitude and longitude is that it is robust to relocation and scaling. More formally, let $\sigma = (\alpha_1, \beta_1), ..., (\alpha_n, \beta_n)$ be a set of positions. Let Σ be the set of traces $\sigma_1, ..., \sigma_m$. Let $min_\alpha(\Sigma) = min\{\alpha \mid \exists \sigma \in \Sigma.(\alpha, \beta) \in \sigma\}$ and $max_\alpha(\Sigma) = max\{\alpha \mid \exists \sigma \in \Sigma.(\alpha, \beta) \in \sigma\}$ and analogously for $min_\beta(\Sigma)$ and $max_\beta(\Sigma)$. Let $norm(\alpha, \Sigma) = \frac{\alpha - min_\alpha(\Sigma)}{max_\alpha(\Sigma) - min_\alpha(\Sigma)}$ and $norm(\beta, \Sigma) = \frac{\beta - min_\beta(\Sigma)}{max_\beta(\Sigma) - min_\beta(\Sigma)}$. Let $\Sigma(\lambda, v, w) = \{\sigma' \mid \sigma \in \Sigma \ \wedge \ |\sigma| = |\sigma'| \ \wedge \ \forall(\alpha, \beta) \in \sigma. \ \exists(\alpha', \beta') \in \sigma'. \ \alpha' = \lambda\alpha + v \wedge \beta' = \lambda\beta + w\}$ be the same set of traces, but scaled by λ and shifted in latitude by v and in longitude by w. Then we have to show the following proposition.

Proposition 4.1. $\forall \sigma \in \Sigma. \ \forall(\alpha, \beta) \in \sigma. \ norm(\alpha, \Sigma) = norm(\lambda\alpha + v, \Sigma(\lambda, v, w)) \wedge norm(\beta, \Sigma) = norm(\lambda\beta + w, \Sigma(\lambda, v, w))$

Proof. Let $\sigma \in \Sigma$ and $(\alpha, \beta) \in \sigma$. Then

$$
\begin{aligned}
norm(\alpha, \Sigma) &= \frac{\alpha - min_\alpha(\Sigma)}{max_\alpha(\Sigma) - min_\alpha(\Sigma)} \\
&= \frac{\lambda(\alpha - min_\alpha(\Sigma))}{\lambda(max_\alpha(\Sigma) - min_\alpha(\Sigma))} \\
&= \frac{\lambda\alpha - \lambda min_\alpha(\Sigma)}{\lambda max_\alpha(\Sigma) - \lambda min_\alpha(\Sigma)} \\
&= \frac{\lambda\alpha + v - (\lambda min_\alpha(\Sigma) + v)}{\lambda max_\alpha(\Sigma) + v - (\lambda min_\alpha(\Sigma) + v)} \\
&= \frac{\lambda\alpha + v - min_\alpha(\Sigma(\lambda, v, w))}{max_\alpha(\Sigma(\lambda, v, w)) - min_\alpha(\Sigma(\lambda, v, w))} \quad \text{(linearity of } max_\alpha \text{ and } min_\alpha\text{)} \\
&= norm(\lambda\alpha + v, \Sigma(\lambda, v, w))
\end{aligned}
$$

And analogously for $norm(\beta, \Sigma)$ □

The heading just needs to be divided by 360. We proceed in the same way with pitch and roll, but we have to calculate modulo 360 beforehand. This is because roll may also be negative depending on whether the drone is tilted to the right or left. The same applies to pitch with the difference that the drone tilts forward or backward.

4.2 Anomaly Detection

This section aims to describe our approach to anomaly detection using machine learning. In the previous section we described how we generated the training data. Now the task is to learn a classifier with the help of this data.

4.2.1 Initial Considerations

A brute-force solution to our problem would have been to use only one large multi-label classifier. This means that all data of a flight would have served as input for the classifier. The classifier would have returned for each sample point an anomaly type if applicable. This method has multiple drawbacks. First of all the number of features is very high. In general, a high feature space means that more training data is needed for a satisfactory result. Secondly, the number of possible classes is very high, which means that we need even more training data. Last but not least this is a multi-label classification problem, which is considered as one of the hardest machine learning problems since it is a generalization of multi-class classification problems. Problems of this kind are often separated into smaller problems that are easier to solve.
The question that eventually led us to change our mind was: if we subdivide a problem into smaller problems, as is often the case with multi-label classification problems, then why would we not make a subdivision specially adapted to this problem? Our problem has a characteristic that makes it a simpler problem than we originally thought: It is not relevant where exactly an anomaly occurs in flight, since the characteristics of the anomaly remain the same. With wind the flight direction changes or the pitch and roll of the drone, with speed anomalies the speed is too slow and with GPS anomalies one can see a jump in the position. Only the speed is complicated when the drone makes a turn. However, curves also have clear characteristics, for example that the drone comes to a standstill and changes its direction.
Our approach is now twofold. First we try to find out at which sampling points anomalies occur. After that, these anomalies are extracted and classified. In the following we explain these steps in more detail.

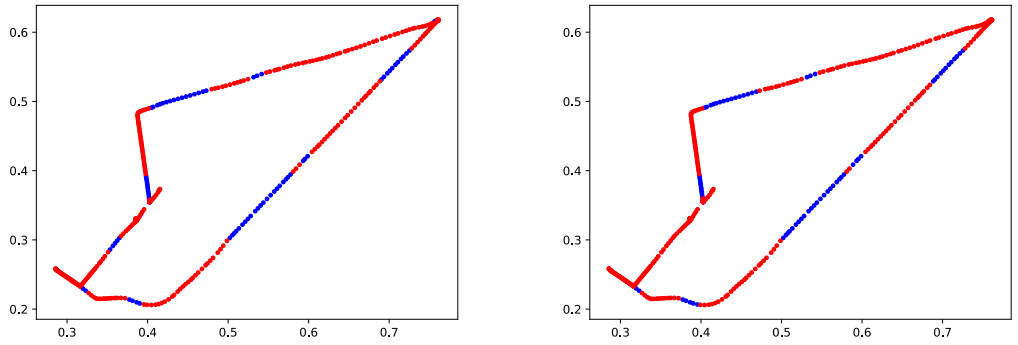

FIGURE 4.8: Example of a classification in step 1 illustrated on position information of the flight. x-direction is the norm of longitude and y-direction is the norm of latitude. Left side is the expected result, right side is the result of the classifier

4.2.2 Anomalous or Not?

The first step is to determine for each individual sampling point whether it is abnormal or not. For this we create a classifier for each sampling point. The features of this classifier are the information contained in the four previous sampling points, those of the following four and those of the point itself. This is necessary because otherwise a GPS anomaly may not be detected as an anomaly, as it only becomes noticeable by a greater distance between two locations. Since we are unable to guarantee a balanced data set, the underrepresented elements in the training data are extended by synthetic data using SMOTE. The result is a stream of 600 values for each flight whose values are either 1 or 0, meaning that the point was either anomalous or not. Figure 4.8 shows an example classification. The red points indicate that this sampling point was abnormal, the blue points that it was not. On the left side is the expected result and on the right side the result of the classifier. The only noticeable difference is at normalized position $(0.35, 0.3)$ where the classifier classified a small part as anomalous, although it is not.

4.2.3 Classification of Anomalies

Prior to classifying anomalies we need to extract them. Hence, we define the following parameters to adjust this process. First of all we need to define a maximum size of an anomaly because the number of features is fixed. On the other hand, it might also be useful to define a minimum size. The reason for this is that extremely short anomalies are very likely no anomalies and may only be misclassified by the classifier in the first step. However, it can also be the case that anomalies are incorrectly classified as normal. For this case we define another buffer parameter which indicates how many points classified as

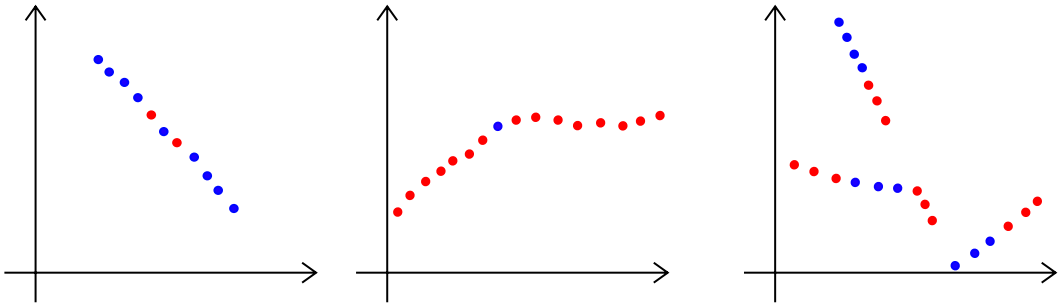

FIGURE 4.9: Examples of anomalous sampling points. The left one is not considered as an anomaly with a buffer of 2 and a minimum size of 5. The middle one is considered as 1 anomaly. The right one is considered as 2 anomalies.

normal may be included in an anomaly. Last, we defined a context size. This parameter determines the length of the sections before and after the anomaly. The idea is to provide the machine learning algorithm with more context for classification. If an anomaly is shorter than the maximum size, the context size is increased such that all anomalies are of the same length.

As an example consider Figure 4.9. Suppose we have a minimum size of 5, a buffer of 2, a maximum size of 20 and a context of 10 sampling points. The red dots represent sampling points that were classified as anomalous in the first step. The blue ones are those that were classified as normal. The first example is not extracted as the size is less than 5. Even with the help of the buffer we only get a total length of 4. The second example is extracted because it is of size 15 and in between there is only one sampling point that is not considered anomalous. In the last example both anomalies are extracted, as they both have a size of 6.

Before we feed our classifier the extracted anomalies, we adjust some values to simplify the classification. Our goal was to superimpose the anomalies, which means that we need to adjust latitude and longitude values. This is achieved in the following way. Suppose we have two streams of latitude and longitude values as follows:

$$lon = lon_1, lon_2, ..., lon_n$$

$$lat = lat_1, lat_2, ..., lat_n$$

First, we set the values relative to the origin starting from the first value as follows:

$$lon = lon_1 - lon_1, lon_2 - lon_1, ..., lon_n - lon_1$$

$$lat = lat_1 - lat_1, lat_2 - lat_1, ..., lat_n - lat_1$$

Now we compute the distance from (lon_n, lat_n) to the origin with the help of the pythagorean theorem: $dist = \sqrt{lat_n^2 + lon_n^2}$ and define a new point $u = (dist, 0)$, which

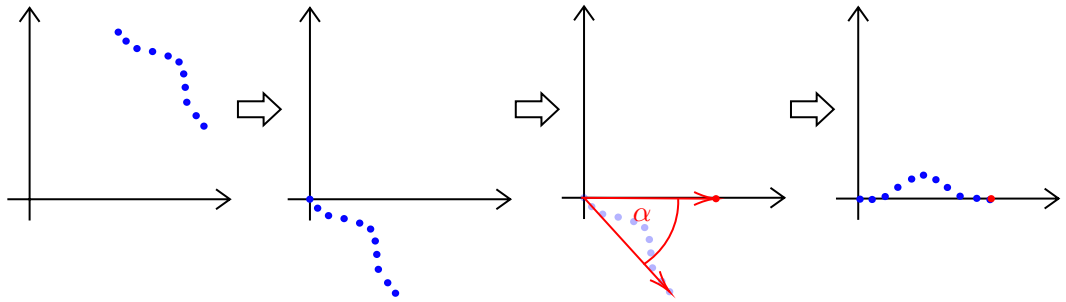

FIGURE 4.10: Illustration of transformation of latitude and longitude. First, we shift the anomaly such that the first sampling point is located on the origin. Then we rotate the anomaly such that the last sampling point is on the x axis.

lies on the x-axis and has the same distance to the origin as $v = (lon_n, lat_n)$. We compute the angle between the two vectors by the formula:

$$angle = acos\left(\frac{u \cdot v}{||u|| \cdot ||v||}\right), \ where \ ||x|| = \sqrt{x_1^2 + x_2^2}$$

Finally, we rotate the remaining points $(lon_2, lat_2), ..., (lon_{n-1}, lat_{n-1})$ by the calculated angle with the help of a rotation matrix. Figure 4.10 illustrates the procedure described above. Apart from the positions we need to adjust the heading, too. This is straightforward, because we only have to rotate it by the angle calculated above.
The creation of the training data is analogous with the difference that the classes are already assigned directly when the anomalies are extracted. The classification problem is now a multi-label classification problem with 3 labels for the 3 possible anomalies. We make use of the label powerset transformation method to transform this problem to a multi-class classification problem with 8 classes. Again, since we cannot guarantee a balanced dataset, SMOTE is used to extend the underrepresented elements in the training data by synthetic data. Figure 4.11 shows an example classification. On the left side there is the expected result and on the right side the classification. One can see that the classifier did not find the GPS anomaly right at the starting point. Furthermore, it often regarded pure wind anomalies as speed anomalies as well.

4.3 Causality Analysis

Our ultimate goal is to present the user with few anomalies that are sufficient for the drone to fail in the end. We define a flight as failed if the battery is empty before it lands or if the drone has not arrived at the target position at all.
Our idea is to estimate the intensity of anomalies with the help of machine learning. In other words, we evaluate the anomalies on a scale from 0 to x, where 0 means that the

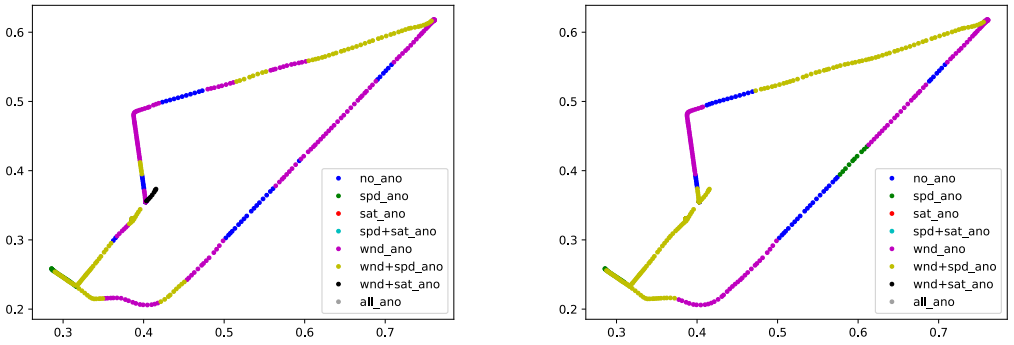

FIGURE 4.11: Example of a classification in step 2 illustrated on position information of the flight. x-direction is the norm of longitude and y-direction is the norm of latitude. Left side is the expected result, right side is the result of the classifier

anomaly had no influence on the flight and x that it had a particularly high influence on it. Then for each type of anomaly a regressor is trained with the anomalies of the respective type. The exact value of x does not matter since the regressors return real numbers. We set $x = 100$. While the scale is the same for each regressor, the different anomalies are not equally severe. Therefore, we create parameters for each type of anomaly whose values we determine with the help of an optimization problem. For this we define an error function that punishes scores on flights that are higher than a certain threshold but did not fail and vice versa. A minimization function then determines the values of the parameters for each anomaly type such that this error function is minimized. Finally, we determine all combinations of anomalies sufficient to cause the flight to fail and present them to the user. In the following subsections we present the individual steps in more detail.

4.3.1 Handcrafting the Intensity of Anomalies

Determining the intensity of an anomaly turned out to be more challenging than originally thought. The intensity depends significantly on what the drone is currently doing. For example, a gust of wind has almost no influence on the drone when it is flying slowly which is the case in curves. Therefore, we decided to always rate wind anomalies occurring in curves with a wind speed of less than $16\frac{m}{s}$ with 0. Speed anomalies are completely ignored there and given a 0 rating, as the drone flies slowly there anyway and an even slower speed has no effect. Apart from that the anomalies are scored as follows.
For the rating of a wind anomaly both wind speed and wind direction, are important. The extent to which the wind direction plays a role was determined with the

help of an experiment. In this experiment the drone flew the route described above 180 times with a 10 times accelerated simulation speed. 9 seconds after the start of the simulation a gust of wind with a speed of $20\frac{m}{s}$ was started, which lasted 6 seconds and blew exactly against the drone. With every flight the angle of the gust of wind was changed by 1 degree until it blew exactly from behind the drone. During the gust the drone always travelled between the first and the second waypoint. After each flight we measured the state of charge of the battery. The result of this experiment is depicted in Figure 4.12. One can clearly observe that the wind direction is of major importance, as the remaining battery capacity varies between -4% and 15%. We use this result for the scoring and transform it into a number between 0 and 1 as follows. Let *res* be a result of the experiment. Then $i_{w_d} = 1 - \frac{res+4}{19}$ is the impact of the direction of a wind gust. The wind direction in an anomaly is calculated relative to the heading, where 0 degree means the wind comes from the opposite direction of the drone, and 180 degree means the wind comes from the same direction as the drone. The impact of the wind speed is computed similarly. We assume a maximum wind speed of $20\frac{m}{s}$. Let *spd* be the speed of a gust. Then $i_{w_v} = \frac{spd}{20}$ is the impact of the wind speed. The drones heading, wind speed and direction of an extracted anomaly are defined as the average value of the occuring headings, wind speeds and directions within the anomaly. Finally, we need to include the sampling interval. Let l be the length of an anomaly, m the maximum length of an anomaly, s be the sampling interval of an anomaly and t the maximum sampling interval of all flights. Then $i_\ell = \frac{l \cdot s}{m \cdot t}$ is the impact of the length of the anomaly. The impact of a wind anomaly is now calculated as follows. Let x be the maximum possible rating of an anomaly. Then:

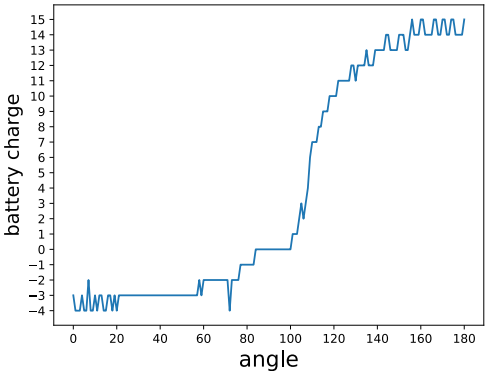

FIGURE 4.12: Battery charge after 180 drone flights with a wind gust whose direction is always shifted by 1 degree.

$$i_w = min(i_{w_v} \cdot i_{w_d} \cdot i_\ell \cdot x, x)$$

is the impact of a wind gust on a scale from 0 to x.

Computing the impact of speed anomalies is a bit easier. The drone has a maximum speed of $15\frac{m}{s}$. Let a be the average speed of the drone during the speed anomaly. Then $i_{s_r} = 1 - \frac{a}{15}$ is the impact of slow speed. The final impact is computed as follows:

$$i_s = min(i_{s_r} \cdot i_\ell \cdot x, x)$$

is the impact of a slow speed on a scale from 0 to x. Another idea was to make use of the reference flight described in Section 4.1.4 instead of just the maximum speed $15\frac{m}{s}$. But the result has not changed for the better, which is why we opted for the simpler approach.
Finally we need to compute the impact of a GPS anomaly. For this, we computed the maximum distance between two consecutive locations within an anomaly. With the GPS glitches described in Section 4.1.2, we achieve a displacement of the drone of at least about 26 meters and at most about 267 meters. Thus, a glitch has the lowest impact if the position shifts by only 26 meters and the highest if it shifts by 267 meters. With a small buffer we compute the impact of a GPS anomaly as follows. Let $m > 25$ be the largest jump in positions. Then $i_{g_r} = \frac{m-25}{245}$ is the impact of a GPS anomaly. The final impact is computed as follows:

$$i_g = min(i_{g_r} \cdot x, x)$$

is the impact of a GPS glitch on a scale from 0 to x. Note that the length of the anomaly makes no difference in GPS glitches and therefore is not included.

4.3.2 Parameter Optimization

All anomalies are scored on the same scale between 0 and x. However, not all types of anomalies are equally severe. Therefore, we define parameters u, v and w for the respective weighting of speed, wind and GPS anomalies. We define an error function as follows. Let $Flights$ be the set of all flights. Furthermore, let $score(f, u, v, w) = u \cdot sum_{spd}(f) + v \cdot sum_{wnd}(f) + w \cdot sum_{sat}(f)$ the summed score of all anomalies of a flight f weighted by the parameters. Then

$$err_f(u, v, w) = \sum_{f \in Flights} err(score(f, u, v, w), failed(f))$$

$$err(x, failed) = \begin{cases} 0, & x \geq 500 \ \wedge \quad failed \ \vee \ x < 500 \ \wedge \ \neg failed \\ 500 - x, & x < 500 \ \wedge \quad failed \\ x - 500, & x \geq 500 \ \wedge \neg failed \end{cases}.$$

is the error function we want to minimize. Fortunately, with the scipy library [72] python already offers an API to minimize such functions using BFGS. Figure 4.13 shows how well we can separate failed flights from successful flights using scores and optimized parameters. The figure shows 4800 flights. The red dots represent failed flights and the green dots represent successful flights. The threshold that separates failed flights from successful flights is 500. We have an overall precision of 0.91 and a recall of 0.88.

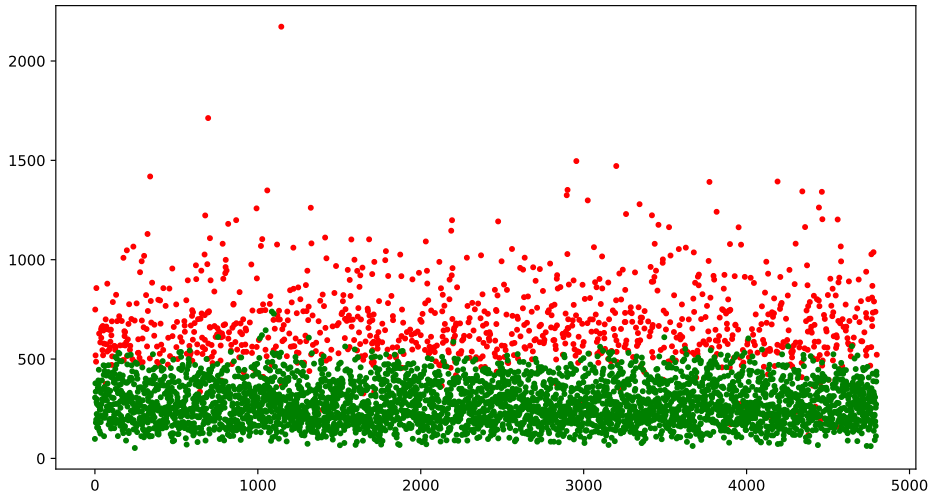

FIGURE 4.13: Summed anomaly scores of 4800 flights. Red dots are failed flights. Green dots are successful flights. x-direction are the flights and y-direction shows the scores.

Thus, the failed flights are not perfectly separated from the successful flights. Especially at the transition from failed to successful flights there are many overlaps. However, a perfect subdivision is not necessary because we want to identify the worst anomalies. Nevertheless, it is a good indicator of how accurate our scores are.

4.3.3 Determination of Causes

Finally, we want to present anomalies to the user that are likely causing the drone to fail in the end. One possibility is to present all possible combinations, which together result in a score of over 500. However, this results in a number of possible combinations that is far greater than the number a human can still handle. Therefore, we need to make some adjustments to significantly reduce the number of possible combinations.
A possible improvement is to re-assemble anomalies of the same type that occur in direct succession and add up the scores. Since our extracted anomalies have a fixed length, it is possible that long lasting anomalies are split into several parts. These will now be re-assembled. A second improvement is to present only minimal subsets. A minimal subset X is a set such that $\forall x \in X$ the sum of all scores in $X \setminus \{x\}$ is less than 500. Although this brought a significant improvement, the number of possible combinations was still too high. Our final approach was therefore to present the most serious anomalies, sufficient to achieve a score of more than 500. Figure 4.14 shows the result for our example flight. On the left side is again the expected result and on the right side is the result of our approach. One can see that our approach has identified almost all anomalies as serious,

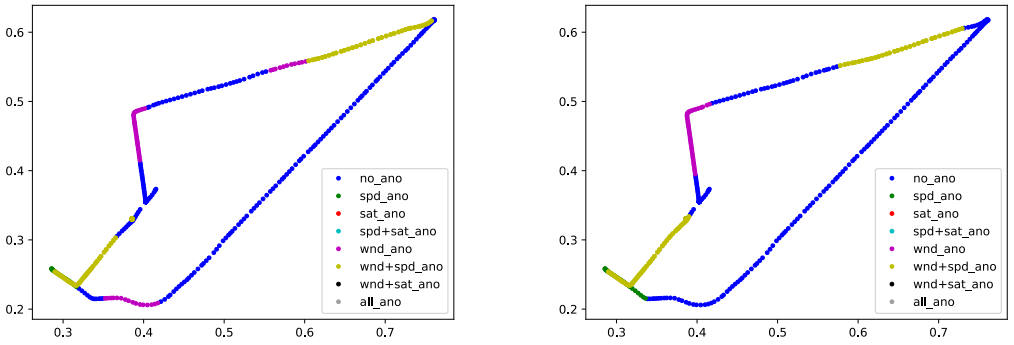

FIGURE 4.14: Most severe anomalies that are sufficient on their own to get a summed score of over 500. x-direction is the norm of longitude and y-direction is the norm of latitude.

which are also considered severe by the reference. Only one wind anomaly in the lower right was not considered severe by the corresponding regressor.

Chapter 5

Evaluation

The evaluation is divided into three parts. First, we want to find out which classifier or regressor is best suited for each step. For this we cross-validate all classifiers and regressors mentioned in the background with different parameters. The best models are then used to perform a final analysis of our approach. Finally we test the anomaly detection on a rotated and relocated mission to show how this little change affects the result.

5.1 Finding suitable classifiers and regressors

All presented classifiers and regressors have different parameters for perfect adaptation to a problem. However, choosing the right parameters is not trivial. A typical approach is to test the models for different parameters known as grid search. In the following we will briefly discuss which parameters we have checked for each classifier and regressor.

k-Nearest-Neighbor

k-Nearest-Neighbor was checked for $k \in \{5, 8, 11, 14\}$ and euclidean distance. All neighbors are equally weighted. These parameters are the same for classification and regression.

Random Forests

For random forests we have varied the number of decision trees. We have tested this method for $60, 90, 120$ and 150 decision trees. Additionally, both split criteria, gini and entropy, have been tested. The trees are expanded until all leaves contain only elements of a single class. The number of features considered at each node is $\sqrt{|T|}$, where T is

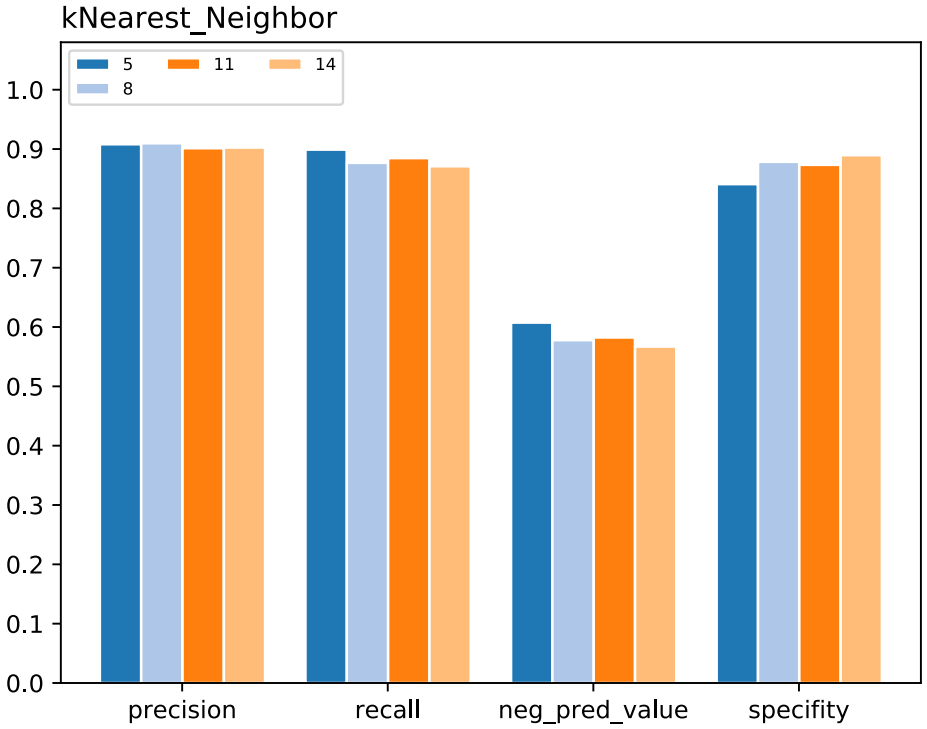

FIGURE 5.1: Results for step 1 for k-Nearest-Neighbor for $k \in \{5, 11, 8, 14\}$

the training set. The same parameters are used for classification and regression, except for the impurity which we will only check for mean squared error in regression.

Multi-layer Perceptron

For neural networks we used hidden layer sizes of $60, 90, 120$ and 150 respectively with minimizing functions stochastic gradient descent and LBFGS. The activation function of the neurons is chosen to be the logistic sigmoid function. We test three different values for α for the L2 regularization, namely $0.0001, 0.01$ and 10. For stochastic gradient descent we decided for a constant learning rate $\gamma_i = 0.001\ \forall i$ and a momentum $\eta = 0.9$. This should guarantee us fast steps in the beginning, which get smaller the closer we get to the minimum. The number of samples used per optimization round is set to 200. Both stochastic gradient descent and LBFGS stop optimizing as soon as the loss has not improved by at least 10^{-4} after 10 consecutive iterations. In addition to that, the optimization is always aborted after at most 200 iterations. The parameters for both, classification and regression, are the same in our setting.

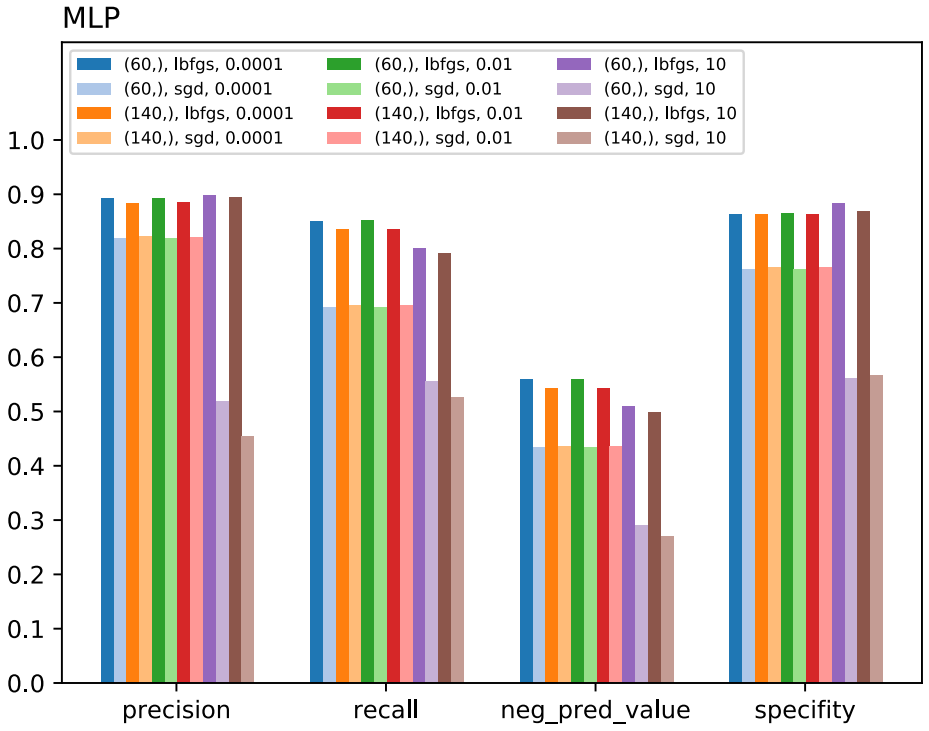

FIGURE 5.2: Result for step 1 for the Multi-layer Perceptron. First value is the number of neurons in the hidden layer. Second is the minimization function and third value is the α

5.1.1 Anomaly Detection

For the evaluation of step 1 we consider a quantity of 15,000 flights. Thus, each of the 600 classifiers is evaluated on 15,000 samples. Each model is evaluated with the help of a stratified 5-fold cross-validation. The training data was each time extended by synthetic data using SMOTE. Figure 5.1 shows the results for different k in k-Nearest-Neighbor, averaged over the mean value of the 5-fold cross-validation of the 600 classifiers. One can see that this approach shows significant weaknesses in the negative predictive value, no matter which k we choose. This means that many of the sampling points classified as non-anomalous are actually anomalous. In this case k-Nearest-Neighbor is hardly better than a random generator. However, the values for precision, recall and specificity are quite good. If a sampling point is not anomalous, this will be detected in 90% of cases. This also applies vice versa: if a sampling point is anomalous, this is also detected in 90% of cases. And finally, 90% of the anomalous sampling points are actually anomalous. The difference between recall and negative predictive value indicates that we have significantly more anomalous sampling points than non-anomalous ones.

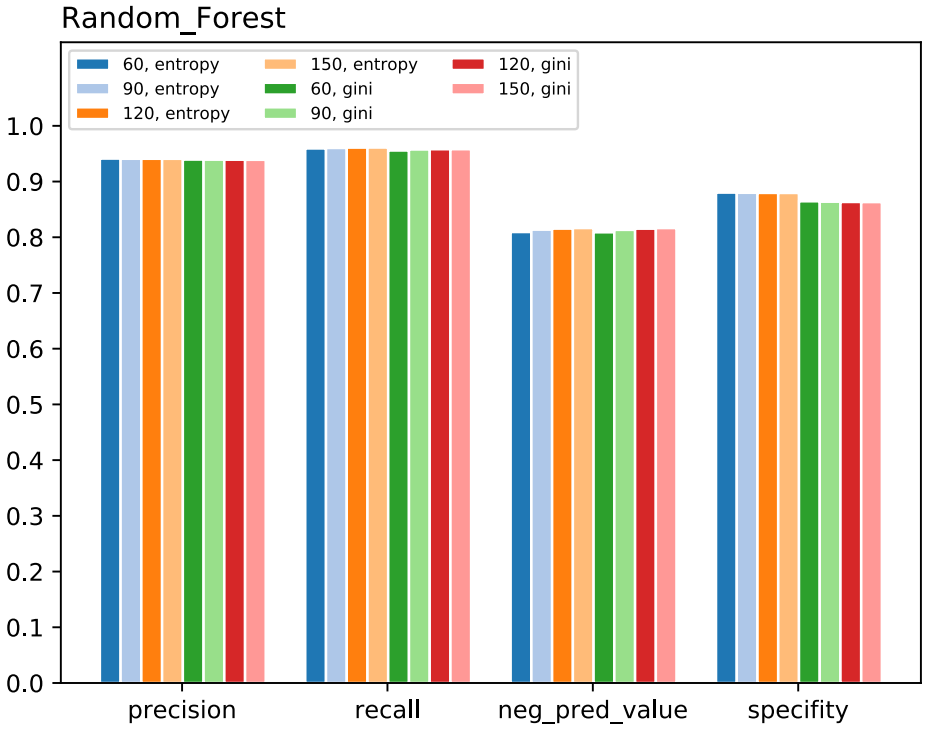

FIGURE 5.3: Result for step 1 for Random Forests for different numbers of decision trees (60, 90, 120, 150) and two different impurity functions, gini and entropy.

Figure 5.2 shows the results for the neural network which are significantly worse than the ones from k-Nearest-Neighbor for some of the parameters. First of all, it is noticeable that the classifiers optimized using the stochastic gradient descent perform worse, which becomes particularly apparent with high α for the L2-regularization. Here, it is probably necessary to re-adjust the parameters for step size and momentum. The approaches optimized using LBFGS perform significantly better, but not better than k-Nearest-Neighbor. One can see slightly better results with a smaller hidden layer, so it might be helpful to further reduce the complexity of the network. Maybe the training data was just too small to train the neural network sufficiently. Unfortunately, up until now we were unable

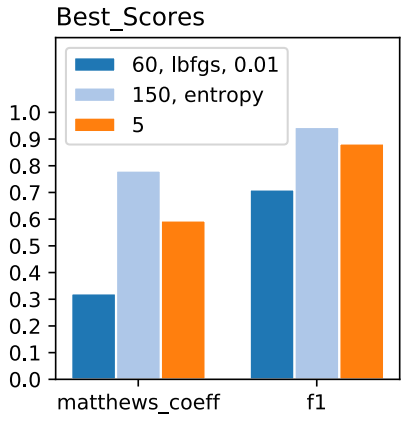

FIGURE 5.4: MCC and F1 score of the best models of kNN, Random Forest and MLP respectively

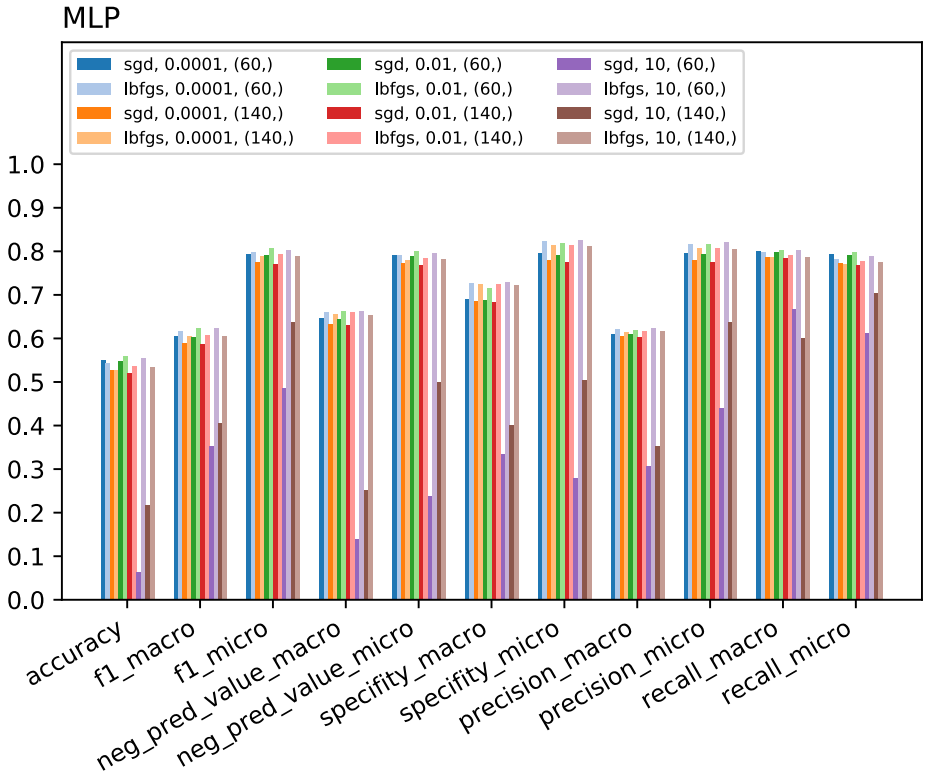

FIGURE 5.5: Scores for Multilayer Perceptron with different parameters for step 2

to find a suitable network that could solve our problem sufficiently well. The results for different parameters of random forest are in contrast quite pleasing, as one can see in Figure 5.3. Although the negative predictive value is still clearly worse than the other metrics, it is still significantly better than with the other models. Apart from that one can see a slightly better result with entropy. The number of decision trees also has a positive effect on the result. The fact that random forest scores much better in every respect than the other models makes the decision for the right model a simple one. The comparison of the Matthews correlation coefficient and the F1 score of the best models of the respective category in Figure 5.4 emphasizes the superiority of random forest once again. While the average MCC value for the best neural network is slightly above 0.3, it is about 0.6 for k-Nearest-Neighbor and almost 0.8 for random forest. Also the F1 score with a value of about 0.95 is much better with the random forest than with the other models.

5.1.2 Anomaly Type Identification

For the evaluation of step 2 we consider a quantity of 86,458 anomalies from 2,000 flights. The anomalies are extracted in sizes of 10. The context size is set to 10 and the buffer

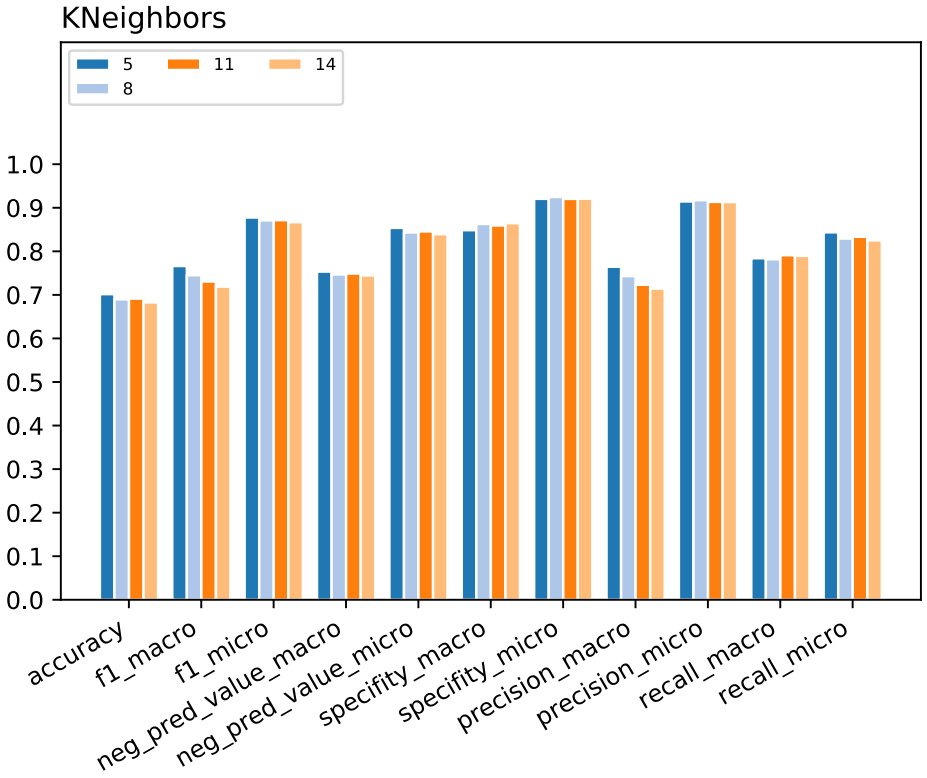

FIGURE 5.6: Scores for k-Nearest-Neighbor with different parameters for step 2

is set to 0. Furthermore, no minimum size of the anomaly is specified, i.e. all anomalies are considered. Each value is the mean value of a stratified 5-fold cross validation. The training data was each time extended by synthetic data using SMOTE. The results for the multilayer perceptron for different parameters are depicted in Figure 5.5. They show that the neural network is the least convincing model. The overall accuracy is just about 57% in the best case and less than 10% in the worst case. However, note that accuracy is a very strict evaluation method, since it only considers an input to be correctly classified if all labels have been correctly classified. Nevertheless, the result is not convincing, since even the less strict F1 scores are only just around 80%, micro-averaged and about 60% macro-averaged. The large discrepancy between micro and macro average values indicates that elements of a less common class are poorly classified in this case. Since GPS anomalies are less common, they are probably the cause for the discrepancy. The overall weaker performance of stochastic gradient descent indicates that the parameters for this procedure were not chosen optimally. Furthermore, one can see that smaller networks perform slightly better in comparison. Thus, it might again be worth considering reducing the complexity of the network.

Surprisingly, k-Nearest-Neighbor is significantly more convincing as one can see in Figure 5.6. One can see that the accuracy reaches almost 70%. However, the number

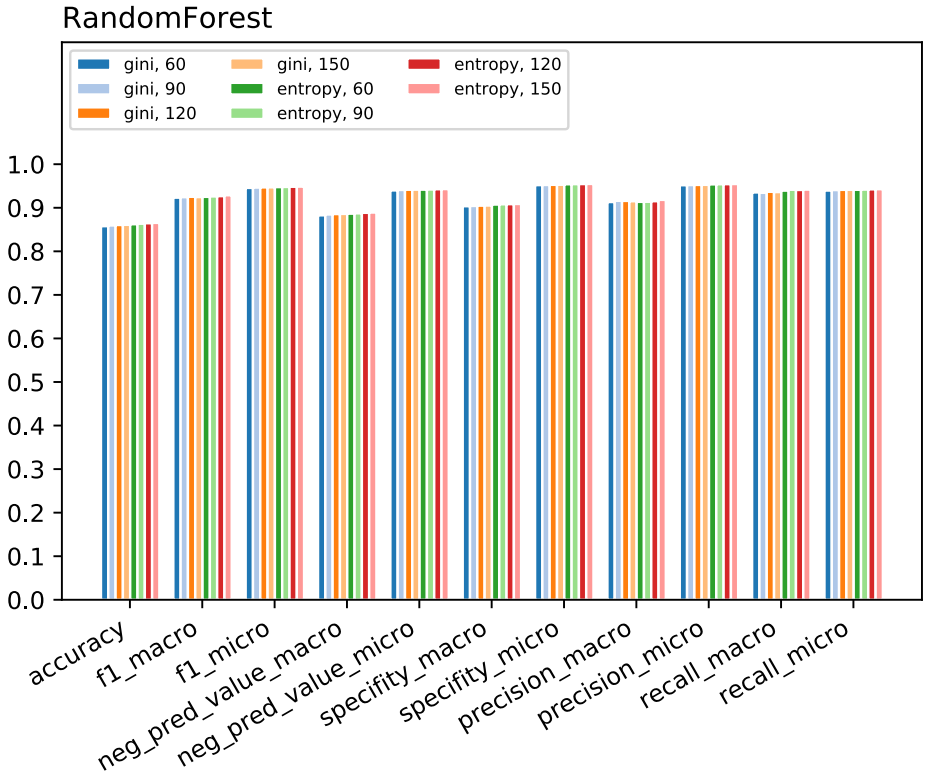

FIGURE 5.7: Scores for random forest with different parameters for step 2

of neighbours hardly affected the result which is why this model probably does not have much room for improvement. Again there is a significant discrepancy between micro and macro average values for the same reason. Especially the discrepancy of precision micro and macro average catches the eye. It indicates that possibly many anomalies were wrongly classified as GPS anomalies. Nevertheless, altogether the result of k-Nearest-Neighbor is acceptable, if one considers the simplicity of the algorithm.

The best result by far was achieved by random forests as one can see in Figure 5.7. With an accuracy of over 80%, this result is pretty good. The discrepancy between micro and macro average values is also much smaller with random forests, but still present. The difference in negative predictive value could indicate that some GPS anomalies were not detected. The very small difference in the results for different parameters indicates that further improvements are difficult to achieve here. Nevertheless, that does not change the fact that the result is already very good.

Figure 5.8 compares the three models with the best parameters. Here again the clearly better result of random forest becomes apparent. But even the results of k-Nearest-Neighbor can sometimes keep up. However, the neural network is far behind. Again one can see that random forests are superior to the other approaches in every respect and are chosen for the final evaluation.

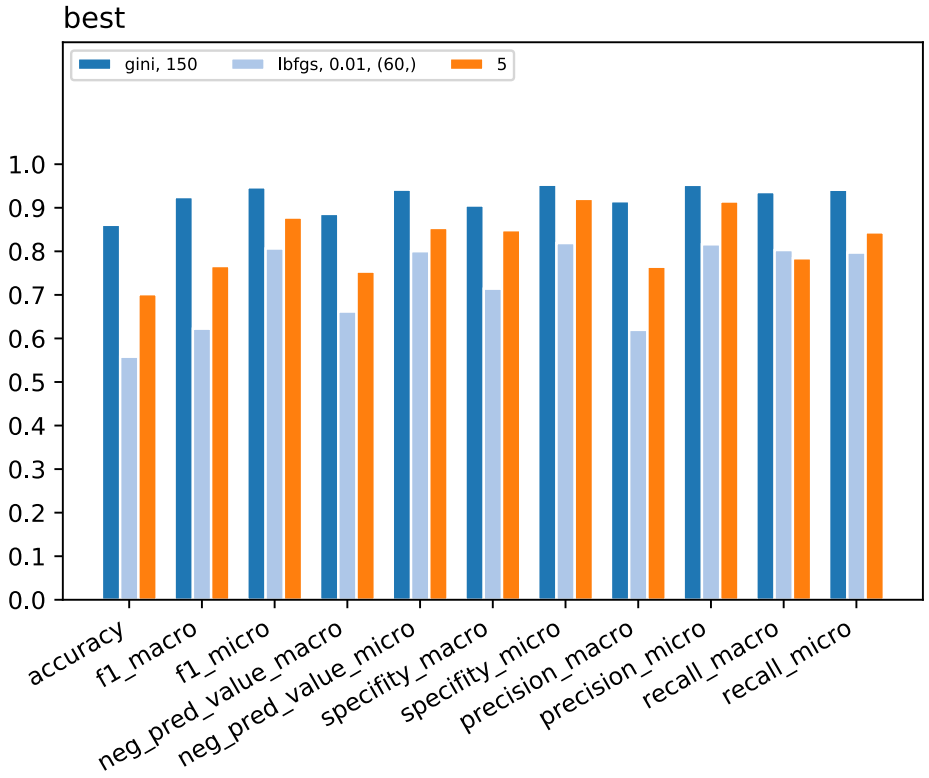

FIGURE 5.8: Comparison of best scores for each model

5.1.3 Anomaly Scoring

For the evaluation of step 3 we considered the same quantity of 86,458 anomalies of the same 2,000 flights as in step 2. Once again each model is evaluated with the help of a stratified 5-fold cross-validation. The results for the speed regressors are depicted in Figure 5.9. The graphic shows the results of the error functions explained in Section 2.2.4 on a logarithmic scale. In contrast to the score functions shown in the previous sections, a lower value is better here. One can see that k-Nearest-Neighbor is the worst overall in comparison. Although the mean squared error and the maximum error decrease with increasing k, the mean absolute error and the median absolute error increase slightly. Nevertheless it might be interesting to check the performance for even higher k since we prefer a minimization of the mean squared error. A slightly better result was achieved by the neural network, with the exception of the case with high complexity and high α. It is interesting to observe that stochastic gradient descent and LBFGS behave exactly the other way around this time. With stochastic gradient descent we achieve slightly better results than with LBFGS. The size of the network does not show any correlation. Although the errors increase significantly with high network complexity and high α, this is not the case with lower α. Here, in fact, the more complex network performs better.

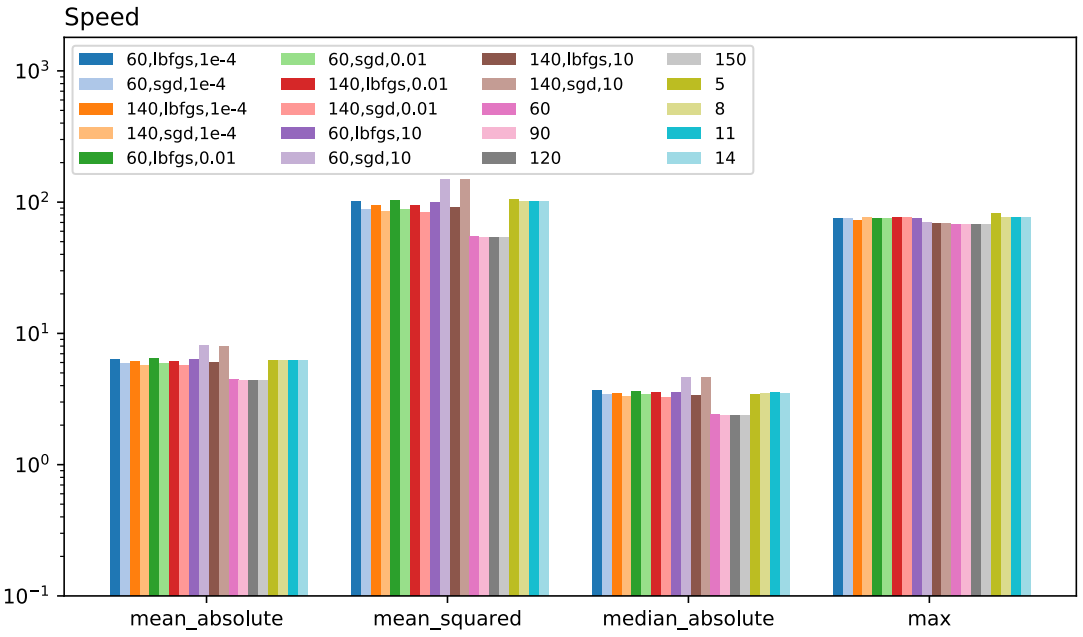

FIGURE 5.9: Different error functions for the different models with different parameters for the regression of speed

The best performance was once again achieved by the random forests. However, the number of decision trees hardly changes the result which could indicate that performance cannot be improved significantly. It is noticeable that with all approaches the maximum error is very high. However, by looking at the median absolute error and the mean absolute error, one can see that such a high error is rather rare. A similar situation can be seen with the regressors for GPS anomalies as depicted in Figure 5.10. Once again, k-Nearest-Neighbor performs worst and random forest performs best. A change in the number of neighbours or the number of decision trees causes a hardly visible change in the result again. Interestingly enough, the neural network performs best with high network complexity and high α which is in strong contrast to the previous chart. Moreover, a worse performance is no longer recognizable with the LBFGS minimization. The maximum error this time is slightly lower on average than for speed anomalies. However the mean absolute error is on the other hand slightly higher. Last but not least we consider the performance of regressors for wind anomalies in Figure 5.11. Here the neural network shows the worst performance and random forest once again the best performance. k-Nearest-Neighbor is about in the middle. However, the result seems to get worse with higher k. The performance of the network is worse again with LBFGS minimization. As with speed, a significant decrease of the result can be seen here again with high α and high network complexity. Compared to the other two cases, the overall error here is lower. Nevertheless, the maximum error is still very high. Figure 5.11 compares the R^2 score of the best regressors of each type for each anomaly. First of all,

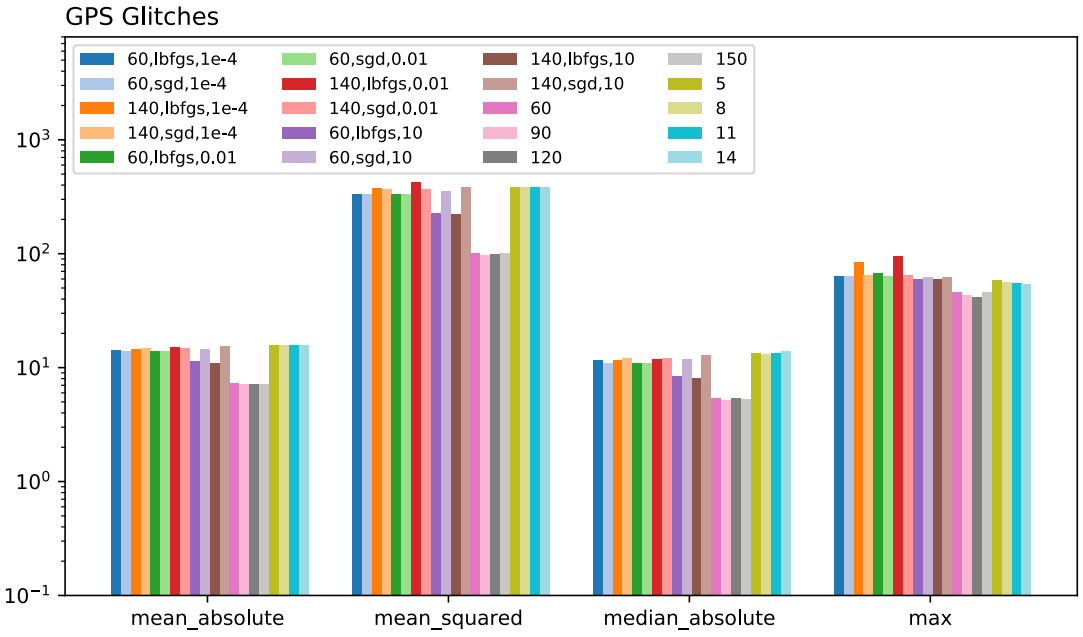

FIGURE 5.10: Different error functions for the different models with different parameters for the regression of GPS glitches

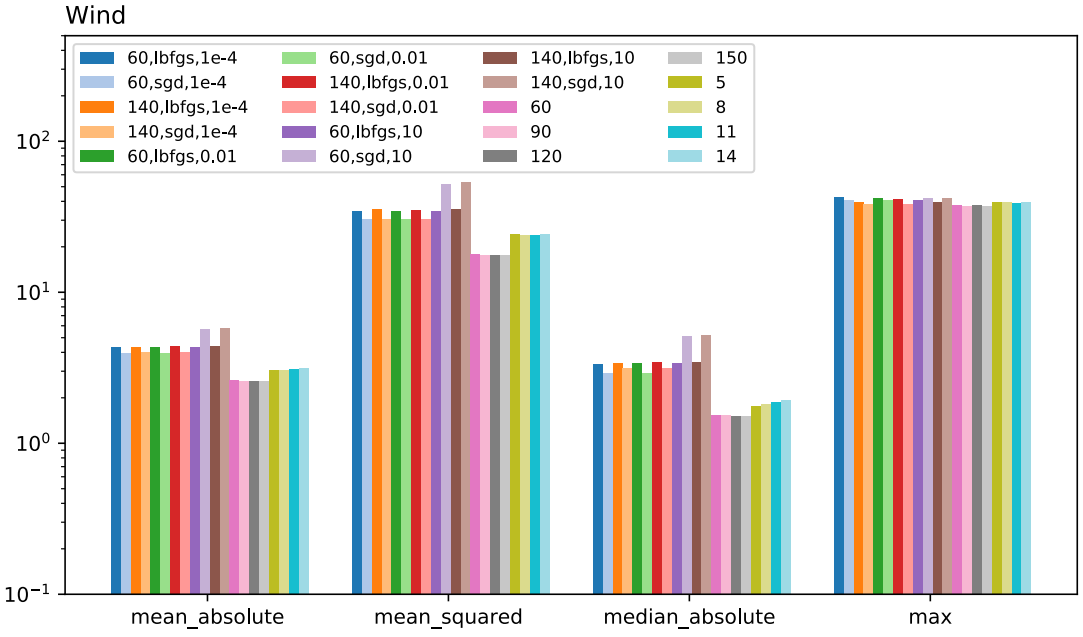

FIGURE 5.11: Different error functions for the different models with different parameters for the regression of wind

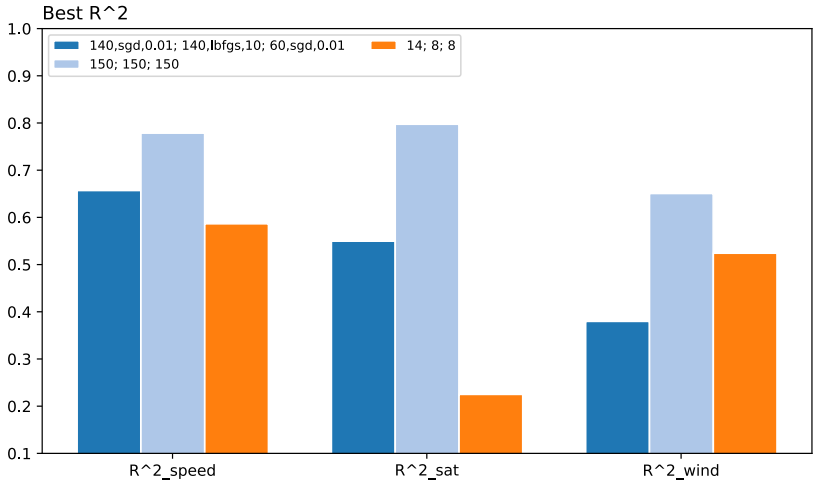

FIGURE 5.12: R^2 scores for the best regressors for each anomaly type

one can see that all regressors achieve better results than a simple regressor that just returns the average value, albeit only slightly in the case of GPS anomalies and k-Nearest-Neighbor. Since k-Nearest-Neighbor only considers neighbors with the least euclidean distance, it is understandable that neighbors with strong jumps in the positions are not as close as for example neighbors with similar speed or pitch and roll. This could explain the poor performance of k-Nearest-Neighbor on GPS anomalies. The performance of random forest is clearly superior to the average regressor, even though it drops on wind anomalies. The underlying training data may not be scored correctly in case of wind anomalies. Since random forest performs better in every aspect than the other two models, the decision for the right regressor is once again easy.

5.2 Evaluation of the Composition

For the evaluation of the composition we consider a total number of 61,205 flights. we take 80% of the data as training set and 20% for evaluation. We make use of the classifiers and regressors that achieved the best results in the previous evaluation. The results of the previous steps will now serve as input for the succeeding steps. We start off with step 1. Table 5.1 shows the absolute results of all sampling points of the remaining

TABLE 5.1: confusion matrix for step 1

Correct \ Predicted	Anomalous	Normal
Anomalous	4,862,739	124,128
Normal	89,574	2,268,159

12,241 flights. It describes the sum of the results of all 600 classifiers. According to this

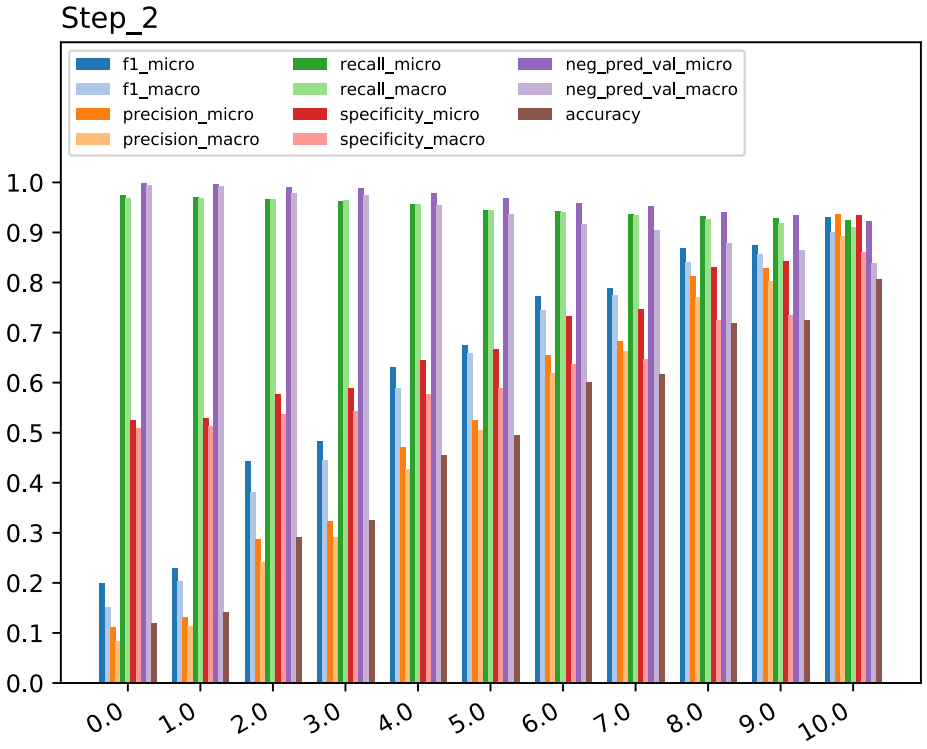

FIGURE 5.13: Result of step 2 for different tolerance values and anomaly size 10

table the overall precision is about 98% and recall is about 97%. On the other hand the negative predictive value is about 95% and specificity 96%. Matthews correlation coefficient results in 0.93. Averaging over the results of the single classifiers results in a slightly worse overall result here the precision decreases to about 95%. Recall remains at about 97%. Specificity drops slightly to 92%. However, the negative predictive value drops sharply to only about 84%. For this reason the Matthews correlation coefficient drops to only about 0.84. This is mainly due to the classifications at the beginning and end of a flight. There are very few anomalies and the classifiers do not have enough data to recognize the small number of anomalies as anomalous. All in all, the result is consistent with the result in the previous evaluation. The higher number of flights has improved the result, but not significantly. Nevertheless, the result is very good and minor errors can still get fixed in step 2 with the help of the buffer.

The evaluation of step 2 based on the data of step 1 is a bit more involved. We first have to define when a classification is correct and when it is not. We say that a classification of a segment is correct if there is a manually classified segment nearby with the same label. More formally, let $s_p, t_p \in \{1, ..., 600\}$ be the starting point and ending point of a predicted anomaly in a flight with 600 sampling points. Given a tolerance value ε, we say that a label l is correct for (s_p, t_p), iff there is an anomaly (s_c, t_c) in the correct

classification with the same label l such that $|t_p - t_c| + |s_p - s_c| < \varepsilon$. The anomalies are extracted in sizes of 10. The context size is set to 10 and the buffer is set to 5. The minimum size of an anomaly is 0. Figure 5.13 shows the result for different values for the tolerance ε. One can see that the result is pretty bad for a tolerance of 0. However, this scenario is quite unrealistic. It is unlikely that the predicted anomalies will start and stop at exactly the same sampling point as the correct anomalies. Besides, there is no need for that to be the case. A slight deviation at the start and end of an anomaly is not significant as long as it is at least detected. The recall value is nevertheless very high. This means that if the anomalies overlap exactly, then the type was correctly predicted most of the time. On the other hand, the negative predictive value is also very high. This is due to the fact that most correct anomalies do not have a type as only very few anomalies are found at a tolerance of 0. With increasing tolerance the other values also increase. With a tolerance of 10 we get pretty good values for all measures above 80%. We consider a tolerance of 10 to be justifiable, since a sufficiently precise determination of the position of the anomaly is still given. Table 5.2 shows the underlying data for the

TABLE 5.2: confusion matrices for step 2 with tolerance $t = 10$ and anomaly size 10

Correct \ Pred.	Speed	No Speed
Speed	325,334	25,480
No Speed	16,172	119,828

Correct \ Pred.	GPS	No GPS
GPS	6,227	837
No GPS	1,558	478,192

Correct \ Pred.	Wind	No Wind
Wind	358,036	30,434
No Wind	29,583	68,761

results with a tolerance of 10. One can see that all types were classified similarly well. To verify the effect of the chosen size of the anomaly, we checked step 2 with anomaly size 20 as well. Context size, buffer and minimum size remain the same. The result is depicted in Figure 5.14. First of all, it is noticeable that we had to increase the tolerance in order to achieve similar results as with the smaller anomaly size. This makes sense because the anomalies can now be further away from each other. However, with a tolerance value of 15, the results are still worse than with an anomaly size of 10 and a tolerance value of 10. This becomes particularly clear with the accuracy, which is now only around 70%. But also specificity dropped to about 74% which means the number of false positives has increased. Table 5.3 shows the results in detail for a tolerance of 15. One can see that all results have become significantly worse. Although there are fewer anomalies overall, the number of misclassifications is very often even higher than with the previous settings. It is particularly noticeable that only half of all non-wind anomalies were detected as such. In general, it can be said that the smaller anomaly size provides a more precise result not least because of the lower tolerance value.

For the evaluation of step 3 we proceed in a similar way. We take the score of the

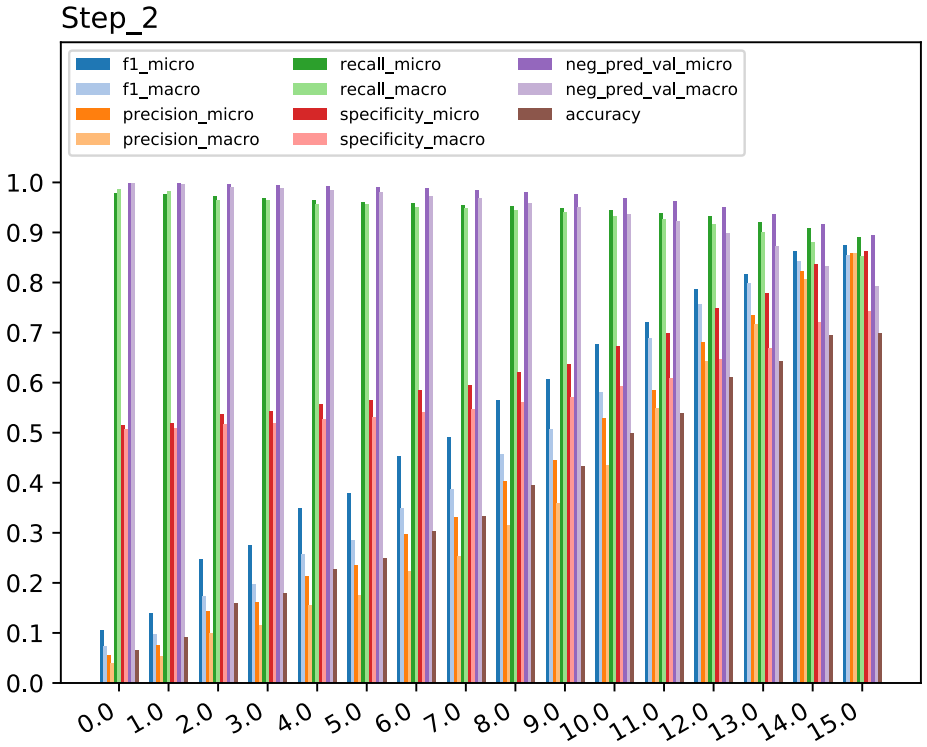

FIGURE 5.14: Result of step 2 for different tolerance values and anomaly size 20

TABLE 5.3: confusion matrices for step 2 with tolerance $t = 10$ and anomaly size 20

Correct \ Pred.	Speed	No Speed
Speed	159,372	23,622
No Speed	22,751	66,487

Correct \ Pred.	GPS	No GPS
GPS	4,234	1,249
No GPS	712	266,037

Correct \ Pred.	Wind	No Wind
Wind	186,935	18,031
No Wind	34,701	32,565

anomaly that is closest and still within the tolerance. If there is none nearby the score is set to 0. Figure 5.15 shows the R^2 scores for the three regressors for different tolerance values and an anomaly size of 10. For values less than or equal to 4, all regressors are worse than the average baseline. Probably no anomaly is found nearby and therefore many scores are set to 0. With a tolerance of 10 we get very close to the results in the first step. Thus one can say that the errors in the previous steps worsen the results of the subsequent steps, but not in such a way that the approach becomes unusable. It is interesting to know how the score distribution is on the failed and successful flights. The result of the classifiers shows that they are cautious in assigning higher scores. About 69% of all failed flights get a total summed score of more than 500. However, 95% of all

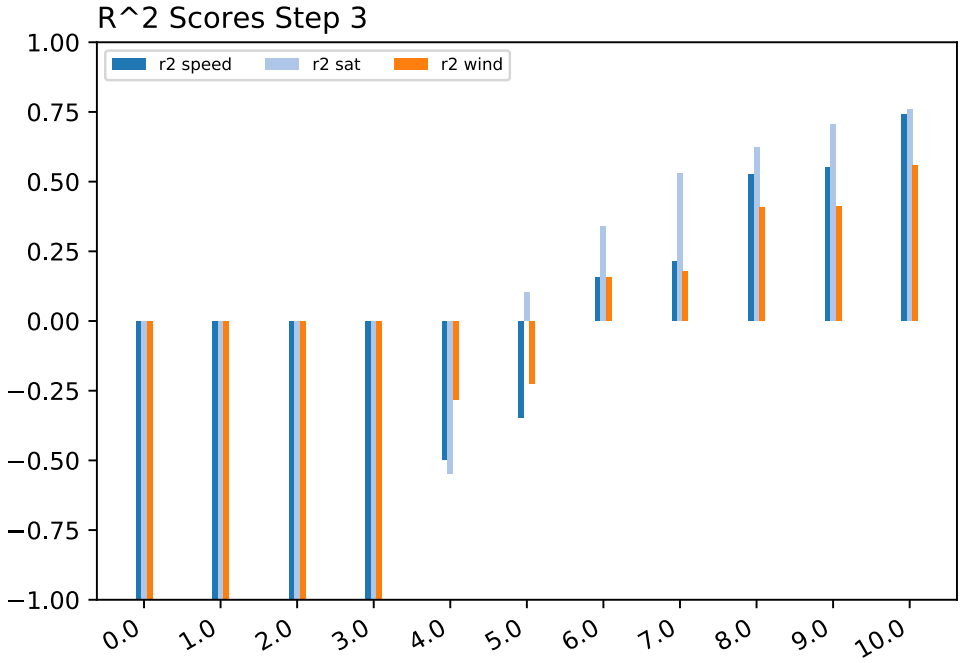

FIGURE 5.15: Result of step 3 for different tolerance values and anomaly size 10

flights with a summed scoring of over 500 are also failed flights. In comparison, for our manually generated scores 83% of the failed flights had a summed score of over 500 and 76% of the summed scores with a value of over 500 were failed flights.

The results for an anomaly size of 20 can be seen in Figure 5.16. Here the result is significantly worse, especially in the case of speed anomalies, despite a higher tolerance value. Up to a tolerance of 10 they are even worse than the average base regressor. It becomes clear once again that with smaller anomaly sizes significantly better results can be achieved. The summed scores can no longer sufficiently separate the failed flights from the successful flights at an anomaly size of 20. Only about 34% of all failed flights also get a summed score of over 500. Though, 99% of all flights with a score of over 500 are also failed flights. Nevertheless, the result is significantly worse for this anomaly size. Again, in comparison, for our manually generated scores 75% of the failed flights had a summed score of over 500 and 83% of the summed scores with a value of over 500 were failed flights.

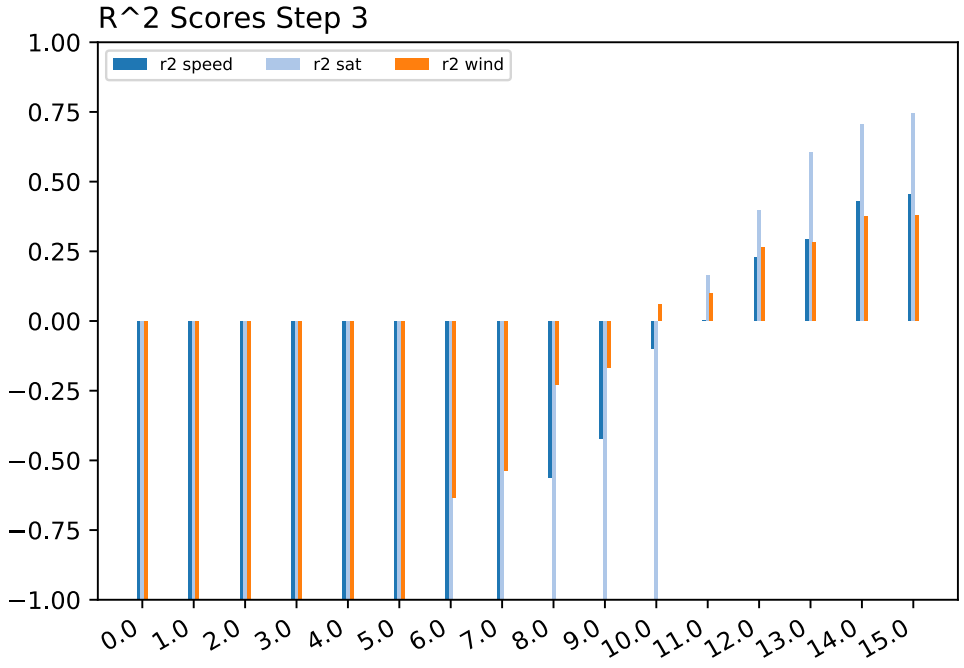

FIGURE 5.16: Result of step 3 for different tolerance values and anomaly size 20

5.3 Rotated and Relocated Mission

While steps two and three can in principle be applied to any possible mission, this is not the case with step 1. In step 1, the classifiers are created for exactly one specific mission. We show on a relocated and rotated mission, that even this small change worsens the result significantly. Our new mission starts at Mount Sunday[1] and takes the same route as the previous one, but rotated 90 degrees. the training data are the same as in the evaluation of the composition. The test data are 2000 flights on the new mission. The result is depicted in Table 5.4. This results in an overall precision of about

TABLE 5.4: confusion matrix for step 1 for a rotated and relocated mission

Correct \ Predicted	Anomalous	Normal
Anomalous	658,858	158,693
Normal	199,959	138,090

76% and a recall of about 80%. The negative predictive value is now only at about 46%. Specificity is even worse and only at about 40%. This results in a poor Matthews

[1]Mount Sunday served as a location for the Lord of the Rings trilogy by Peter Jackson [73], based on the books of J.R.R. Tolkien [74]. It was the setting for Edoras the capital of Rohan.

correlation coefficient of just 0.22. Averaging over the results of the single classifiers the values get even worse. Precision drops to about 74 %, recall to 80%. Specificity drops to 34% and the negative predictive value is slightly better with 63%. Matthews correlation coefficient therefore results in about 0.2. The results clearly show that the approach is not robust against changes in the mission. Although one could make the normalization rotation-independent, as we did in step 2, we could still not use new routes.

Chapter 6

Limitations & Future Work

This work is a first approach towards identifying the causes of failures of system runs, in our case drone flights. In the evaluation we discussed the overall approach and our results. However, there remain issues which can be tackled in future work. In the following we describe disadvantages and limitations of our approach and outline possible solutions.

Anomaly Detection

One of the biggest drawbacks is the great amount of classifiers we have to deal with in the first step. The problem lies in the fact that a high number of classifiers leads to high computational complexity and consequently a long computing time. While the computation can be parallelized, it still requires many CPU cores to achieve quick results. Apart from that, this approach does not scale well. This is because the number of classifiers is constant. Thus, for longer system runs the sampling rate increases, yielding less accurate results. Here we find ourselves in a trade-off scenario, because if more classifiers were to be added, this would in turn lead to an increased runtime. Another limitation is that the classifiers specialize only in one specific run of the system. In our case this corresponds to a fixed drone flight. As an alternative to many classifiers, an approach with a single classifier would also be conceivable. Such a classifier would than distinguish between normal and abnormal behavior of the system and no longer have to specialize in just one particular mission. However, this would require much more training data and was therefore not feasible for the scope of this work. Another option would be to replace the first step with a combination of a runtime verification tool for rough fault location and our approach to Failure Analysis. The purpose of this work, however, was to create a purely machine learning based analysis. Finally, another solution for this could lie in the methods that are also used for natural language processing (NLP) [75]. We see the parallels between NLP and our approach in the fact that we do not know how big the input is, neither for sentences nor for system runs. In order to deal with this

problem, NLP offers, among other things, the approach of recurrent neural networks[1] (RNN). The idea is that the network receives the input sentence word for word and "stores" the information of the sentence in the process. Such an approach would allow us to store more information about the flight in the network in order to make more accurate decisions for each sampling point.

Anomaly Type Identification

A further limitation is that there may be anomaly types whose correct classification would require consideration of the entire lifetime of the anomaly. For our drone example, this is not the case with the anomaly types we are looking at, as they have distinct characteristic differences that show up directly at the beginning of the anomaly and can be unambiguously classified. An increase of the sample points to be considered would, however, make the classification more difficult, since this in turn meant more features for the classifier. Once again one could use natural language processing approaches here, since we do not know the duration of an anomaly in advance. This could solve the aforementioned problem, because we could also "store" information across the lifetime of the anomaly in the RNN.

Anomaly Scoring

Another limitation is the evaluation system to identify the main causes of failed flights. Although we consider only a small number of anomalies, it was not possible to set parameters in a way that would strictly separate the failed flights from the successful ones. There were still enough flights that failed and received a score that was too low and those that did not fail and received a score that was too high. Apart from that, the current approach does not allow for causal chains. This means that anomalies are always regarded as a direct cause of failure. Here, for example, a further step could be introduced as a solution to establish relationships between anomalies. Again, one may be inspired by natural language processing, where it is important to relate words to other words in the sentence.
On a more general note, it might also be interesting to find out why neural networks have performed rather poorly. Although we have tested various configurations, possible sources of error may still lie in the selection of parameters and network depth. It is also possible that a significantly higher number of training data is required. However, an evaluation with even more data was not possible within the scope of this thesis.

[1] A recurrent neural networks allows connections of neurons with neurons of the same layer or previous layers which makes it a generalization of Elman networks.

Chapter 7

Conclusion

We presented an approach for identifying and analyzing anomalies in a system with the help of machine learning. To evaluate this approach, we chose the scenario of an autonomously flying drone where three different anomalies could occur: a gust of wind, wrong GPS inputs or insufficient speed. The evaluation on this example scenario has shown that the approach is capable of detecting abnormal behavior and filtering out the most relevant anomalies. Under the assumption that the previous steps deliver perfect results, the individual steps are highly convincing. With random forests, step 1 achieves an MCC of 0.84 as the average value of all classifiers for a quantity of $61,205$ flights. If one considers the sum of all confusion matrices an MCC value of 0.93 is reached. Step 2 achieves a subset accuracy of over 85% at a quantity of $86,458$ anomalies with random forests. The macro-averaged values show that there is little discrepancy in the classification performance of underrepresented anomalies. For the same anomalies step 3 achieves an R^2 score of about 0.64 for wind anomalies and slightly below 0.8 for GPS and speed anomalies using random forests.

The composition of all three steps achieves a similarly good result with respect to the weakened assumptions. Here we could still achieve a subset accuracy of more than 80% in step 2 with an anomaly size of 10. A similarly slight decrease can be found in step 3, where we still reach an R^2 score of slightly less than 0.75 for GPS and speed anomalies and about 0.55 for wind anomalies.

The evaluation on a rotated mission has shown that the approach also has significant weaknesses, especially in step 1. These could be eliminated in future work. Overall, it can be said that the approach is capable of delivering useful results that can help a failure analyst investigate the cause of the failure.

List of Figures

List of Tables

Appendix A

Copter default parameters

For a simulated UAV we can define several default values to better adapt it to one's own needs. The default values for our copter are depicted in Figure A.1. We will provide a short description of these values in the following.

- EK2_ENABLE enables or disables the extended Kalman filter to estimate, for example, the current position.
- FRAME_TYPE determines the alignment of the motors. 0 means that they are adjusted like a "+" sign.
- MAG_ENABLE enables or disables the compass.
- FS_THR_ENABLE enables or disables a software based throttle failsafe. 1 means that it returns to launch in such a case.
- BATT_MONITOR enables or disables monitoring of the battery.
- COMPASS_OFS* determines the offset of the compass to its respective axis.
- COMPASS_LEARN enables or disables automatic learning of the previously mentioned offsets.
- FENCE_RADIUS determines a circle around the launch that triggers a specific action if it is exceeded.
- FRAME_CLASS determines the type of copter, which is a quadcopter in this case.
- RC* values are for the remote control, which is not relevant for us.
- FLTMODE* allow flight mode changes for different channel PWMs.
- SUPER_SIMPLE enables or disables the super simple mode, which is not relevant for us.

```
EK2_ENABLE      1
FRAME_TYPE  0
MAG_ENABLE  1
FS_THR_ENABLE   1
BATT_MONITOR    4
COMPASS_LEARN   0
COMPASS_OFS_X   5
COMPASS_OFS_Y   13
COMPASS_OFS_Z   -18
COMPASS_OFS2_X   5
COMPASS_OFS2_Y   13
COMPASS_OFS2_Z   -18
FENCE_RADIUS    150
FRAME_CLASS     1
RC1_MAX         2000.000000
RC1_MIN         1000.000000
RC1_TRIM        1500.000000
RC2_MAX         2000.000000
RC2_MIN         1000.000000
RC2_TRIM        1500.000000
RC3_MAX         2000.000000
RC3_MIN         1000.000000
RC3_TRIM        1500.000000
RC4_MAX         2000.000000
RC4_MIN         1000.000000
RC4_TRIM        1500.000000
RC5_MAX         2000.000000
RC5_MIN         1000.000000
RC5_TRIM        1500.000000
RC6_MAX         2000.000000
RC6_MIN         1000.000000
RC6_TRIM        1500.000000
RC7_MAX         2000.000000
RC7_MIN         1000.000000
RC7_OPTION      7
RC7_TRIM        1500.000000
RC8_MAX         2000.000000
RC8_MIN         1000.000000
RC8_TRIM        1500.000000
FLTMODE1        7
FLTMODE2        9
FLTMODE3        6
FLTMODE4        3
FLTMODE5        5
FLTMODE6    0
SUPER_SIMPLE  0
SIM_GPS_DELAY   1
SIM_ACC_RND     0
SIM_GYR_RND     0
SIM_WIND_SPD    0
SIM_WIND_TURB   0
SIM_BARO_RND    0
SIM_MAG_RND     0
SIM_GPS_GLITCH_X    0
SIM_GPS_GLITCH_Y    0
SIM_GPS_GLITCH_Z    0
INS_ACCOFFS_X   0.001
INS_ACCOFFS_Y   0.001
INS_ACCOFFS_Z   0.001
INS_ACCSCAL_X   1.001
INS_ACCSCAL_Y   1.001
INS_ACCSCAL_Z   1.001
INS_ACC2OFFS_X   0.001
INS_ACC2OFFS_Y   0.001
INS_ACC2OFFS_Z   0.001
INS_ACC2SCAL_X   1.001
INS_ACC2SCAL_Y   1.001
INS_ACC2SCAL_Z   1.001
INS_ACC3OFFS_X   0.000
INS_ACC3OFFS_Y   0.000
INS_ACC3OFFS_Z   0.000
INS_ACC3SCAL_X   1.000
INS_ACC3SCAL_Y   1.000
INS_ACC3SCAL_Z   1.000
MOT_THST_EXPO 0.5
MOT_THST_HOVER  0.36
```

FIGURE A.1: Default parameters for the copter

- SIM* values simulate abnormal behaviour, like GPS glitches or wind gusts.
- INS* values set the accelerometer scalings and offsets.
- MOT_THST_HOVER sets motor thrust needed to hover.
- MOT_THST_EXPO determines the thrust curve of the motor.

Bibliography

[1] Waldemar F Larsen. Fault tree analysis. Technical report, PICATINNY ARSENAL DOVER NJ, 1974.

[2] Anjali Joshi, Steve Vestal, and Pam Binns. Automatic generation of static fault trees from aadl models. In *DSN Workshop on Architecting Dependable Systems*. Springer Berlin, 2007.

[3] Faida Mhenni, Nga Nguyen, and Jean-Yves Choley. Automatic fault tree generation from sysml system models. In *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 715–720. IEEE, 2014.

[4] Ben d'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*, pages 166–174. IEEE, 2005.

[5] Community. Ardupilot documentation, 2019. URL http://ardupilot.org/ardupilot/index.html.

[6] Jordi Munoz and Chris Anderson. Ardupilot 1.0, 2009. URL https://www.rcgroups.com/forums/showthread.php?979372-ArduPilot-an-open-source-autopilot-now-available-(-24-95!).

[7] Andrew Tridgell. First flight of ardupilot on linux, 2014. URL https://diydrones.com/profiles/blogs/first-flight-of-ardupilot-on-linux.

[8] ArduPilot. Partners, 2019. URL http://ardupilot.org/about/Partners.

[9] C Kleijn. Introduction to hardware-in-the-loop simulation. In *Control Lab., 7521 AN Enschede.* 2014.

[10] Community. Sitl simulator (software in the loop), 2019. URL http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[11] Bernard Etkin. *Dynamics of atmospheric flight*. Courier Corporation, 2012.

[12] Community. Mavlink developer guide, 2019. URL https://mavlink.io/en/.

[13] Community. Mavproxy, 2019. URL https://ardupilot.github.io/MAVProxy/html/index.html.

[14] Manaswini Rath, Nandeesha Shekarappa, and Venkatesh Ramachandra. Ground control station for uav, October 18 2007. US Patent App. 11/403,472.

[15] Community. Dronekit python, 2019. URL https://github.com/dronekit/dronekit-python.

[16] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[17] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[18] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3):1–13, 2007.

[19] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 254–269. Springer, 2009.

[20] Padraig Cunningham and Sarah Jane Delany. k-nearest neighbour classifiers. *Multiple Classifier Systems*, 34(8):1–17, 2007.

[21] Ernst Wit and John McClure. *Statistics for microarrays: design, analysis and inference*. John Wiley & Sons, 2004.

[22] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[23] Dan Steinberg and Phillip Colla. Cart: classification and regression trees. *The top ten algorithms in data mining*, 9:179, 2009.

[24] Laura Elena Raileanu and Kilian Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

[25] Community. Decision trees, 2019. URL https://scikit-learn.org/stable/modules/tree.html#mathematical-formulation.

[26] Wei-Yin Loh. Classification and regression tree methods. *Encyclopedia of statistics in quality and reliability*, 1, 2008.

[27] Sandhya Samarasinghe. *Neural networks for applied sciences and engineering: from fundamentals to complex pattern recognition*. Auerbach publications, 2016.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[29] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[30] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

[31] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[32] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[33] Carl T Kelley. *Iterative methods for optimization*. SIAM, 1999.

[34] Charles George Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1): 76–90, 1970.

[35] Roger Fletcher. A new approach to variable metric algorithms. *The computer journal*, 13(3):317–322, 1970.

[36] Donald Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of computation*, 24(109):23–26, 1970.

[37] David F Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

[38] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge & Data Engineering*, (9):1263–1284, 2008.

[39] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[40] Feng Zhang. *Cross-validation and regression analysis in high-dimensional sparse linear models*. Stanford University, 2011.

[41] Mervyn Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[42] Seymour Geisser. The predictive sample reuse method with applications. *Journal of the American statistical Association*, 70(350):320–328, 1975.

[43] David L Olson and Dursun Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[44] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

[45] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405 (2):442–451, 1975.

[46] Min-Ling Zhang and Zhi-Hua Zhou. A review on multi-label learning algorithms. *IEEE transactions on knowledge and data engineering*, 26(8):1819–1837, 2013.

[47] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[48] Peter J Rousseeuw and Christophe Croux. Alternatives to the median absolute deviation. *Journal of the American Statistical association*, 88(424):1273–1283, 1993.

[49] 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. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[50] Piotr Szymański and Tomasz Kajdanowicz. A scikit-based python environment for performing multi-label classification. *arXiv preprint arXiv:1702.01460*, 2017.

[51] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017. URL http://jmlr.org/papers/v18/16-365.html.

[52] Grigorios Tsoumakas and Ioannis Vlahavas. Random k-labelsets: An ensemble method for multilabel classification. In *European conference on machine learning*, pages 406–417. Springer, 2007.

[53] Johannes Fürnkranz, Eyke Hüllermeier, Eneldo Loza Mencía, and Klaus Brinker. Multilabel classification via calibrated label ranking. *Machine learning*, 73(2):133–153, 2008.

[54] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *International Conference on Runtime Verification*, pages 152–168. Springer, 2016.

[55] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. Stream runtime monitoring on uas. In *International Conference on Runtime Verification*, pages 33–49. Springer, 2017.

[56] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[57] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[58] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. Real-time stream-based monitoring. *arXiv preprint arXiv:1711.03829*, 2017.

[59] James J Scutti and William J McBrine. Introduction to failure analysis and prevention. *Materials Park, OH: ASM International, 2002.*, pages 3–23, 2002.

[60] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151. IEEE, 1995.

[61] Richard A DeMillo, Hsin Pan, and Eugene H Spafford. Failure and fault analysis for software debugging. In *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International*, pages 515–521. IEEE, 1997.

[62] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[63] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 467–477. IEEE, 2002.

[64] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200. IEEE, 2014.

[65] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[66] Anup K Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Workshop on Intrusion Detection and Network Monitoring*, volume 51462, pages 1–13, 1999.

[67] Shaohui Wang, Yoann Geoffroy, Gregor Gössler, Oleg Sokolsky, and Insup Lee. A hybrid approach to causality analysis. In *Runtime Verification*, pages 250–265. Springer, 2015.

[68] Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. From probabilistic counterexamples via causality to fault trees. In *International Conference on Computer Safety, Reliability, and Security*, pages 71–84. Springer, 2011.

[69] Gregor Gössler and Jean-Bernard Stefani. Fault ascription in concurrent systems. In *Trustworthy Global Computing*, pages 79–94. Springer, 2015.

[70] Community. Mavlink developer guide, 2019. URL https://mavlink.io/en/file_formats/.

[71] Community. Mission planner home, 2019. URL http://ardupilot.org/planner/.

[72] Eric Jones, Travis Oliphant, and Pearu Peterson. {SciPy}: Open source scientific tools for {Python}. 2014.

[73] Department of Conservation. Mount sunday track, 2019. URL https://www.doc.govt.nz/parks-and-recreation/places-to-go/canterbury/places/hakatere-conservation-park/things-to-do/tracks/mount-sunday-track/.

[74] John Ronald Reuel Tolkien. *The Lord of the Rings: One Volume*. Houghton Mifflin Harcourt, 2012.

[75] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.