Stefan Simon

Generating Custom-Fit Polar Diagrams From Performance Measurements On Sailing Yachts

Master's Thesis

Graz University of Technology

Institute of Engineering and Business Informatics
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Siegfried Vössner

Supervisor: Dipl.-Ing. Clemens Gutschi, BSc
and Dipl.-Ing. Birgit Mösl, BSc

Graz, November 2017

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online:
https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________________ ____________________________________

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ______________________________ ____________________________________

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

During a regatta all boats have the same wind conditions and the winning team is the one most efficiently using power of the wind in order to reach the finish line. The ratio of the wind that can be converted into motion power depends on wind speed and its relative angle to the boat. This relationship can be illustrated using performance polar diagrams. Each type of sail has its own characteristic. The variations range from sails that are tailored for fast downwind but can hardly be used upwind until sails specific for close-hauled sailing. Detailed knowledge of the behavior can increase the actual performance of the crew on a boat.

The currently favored method to forecast boat speed based on wind conditions is the use of so called velocity prediction programs. They rely on mathematical models which are based on measurements of the boat and all its appendages.

This thesis develops an alternative, which allows the generation of polar diagrams on sailing boats without additional effort besides normal sailing. The required measurement devices are already installed on common sailing boats. Over the course of this thesis a software was developed which can be linked to the recorded measurements. It uses data analysis to find a model which best represents the underlying data. A regression method is used which supports modeling with measurement errors of all features and non-linear data. Additional models were implemented to add the possible influence of past measurements within a time series by weighting each data point by an established quality function. Out of all implemented models, techniques from machine learning allow an automated selection.

For the examined data the resulting performance polar diagram has a similar shape as the corresponding velocity prediction program. The software indicates slower boat speeds, which result from the actual application of the boat's potentiality by the crew. So their room for improvement, especially on lower wind speeds, is revealed.

In further research a verification of the developed model with extensive measurement data is planned.

Kurzfassung

Während einer Regatta haben alle Boote die gleichen Windverhältnisse, aber es gewinnt nur jenes Boot, das es schafft die Kraft des Windes am effizientesten umzusetzen. Der Anteil des Windes, der in Vortrieb umgesetzt werden kann, ist abhängig von der Windgeschwindigkeit und dem Windwinkel. Meist wird dieser Zusammenhang in einem Polardiagramm dargestellt, welches je nach Segeltyp sehr unterschiedlich aussehen kann. Es gibt Segel die dafür zugeschnitten sind um sehr schnell vor dem Wind zu laufen aber mit denen man kaum gegen den Wind aufkreuzen kann, bis zu Segeln die dafür gemacht sind hart am Wind zu segeln. Die genaue Kenntnis des Verhaltens kann die Leistung der Crew erheblich verbessern.

Die derzeit verwendete Methode um die erwartete Bootsgeschwindigkeit für gewisse Windbedingungen zu bestimmen sind so genannte Velocity Prediction Programs. Diese basieren auf mathematischen Modellen, denen eine Vermessung des Bootes mit allen Anbauten und Segeln vorausgeht.

Im Zuge dieser Masterarbeit wurde eine Alternative entwickelt, die eine Erstellung von Polardiagrammen ohne Zusatzaufwand zum gewöhnlichen Segeln ermöglicht. Die dafür notwendigen Messgeräte sind üblicherweise schon auf den Booten installiert. Im Verlauf dieser Arbeit wurde eine Software entwickelt, die gespeicherte Messwerte mittels Datenanalyse aufbereitet. Das Ziel ist es ein Modell zu finden, das die zugrundeliegenden Daten angemessen abbildet. Dafür wurde eine Regressionsmethode recherchiert, die eine Modellierung bei Messfehlern in allen Variablen und nichtlinearen Daten unterstützt. Weitere Modelle wurden implementiert um den möglichen Einfluss von vergangenen Messungen in der Zeitreihe zu untersuchen. Das passiert durch eine Gewichtung jedes Datenpunktes mit einer dafür eingeführten Qualitätsfunktion. Aus allen implementierten Modellen kann durch Techniken aus Machine Learning das Beste automatisiert ausgewählt werden.

Das resultierende Polardiagramm hat eine gleichartige Form wie die entsprechenden Velocity Prediction Programs. Die Software zeigt im Allgemeinen niedrigere Bootsgeschwindigkeiten an, da die Anwendung des Leistungsvermögens aufgezeigt wird.

Hier lässt sich vor allem bei niedrigeren Bootsgeschwindigkeiten ein Verbesserungspotential der Crew erkennen.

Weiterführend ist eine Verifizierung der entwickelten Modellierung mit umfangreichen Messdaten geplant.

Contents

Abbreviations

ANN	artificial neuronal network	94
API	application programming interface	
AWA	apparent wind angle	61
AWS	apparent wind speed	61
BC	before Christ	
BSD	Berkeley Software Distribution	54
BSP	boat speed	97
BTV	boat velocity	
CLI	command line interface	47
COM	communication port	
CSV	comma-separated values	
DGPS	Differential GPS	42
GPS	Global Positioning System	42
IMS	International Measurement System	1
NMEA	National Marine Electronics Association	94
MIT	Massachusetts Institute of Technology	55
ODR	orthogonal distance regression	95
OLS	ordinary least squares	27
ORC	Offshore Racing Congress	94
RPC	Round Palagruža Cannonball	60
TWA	true wind angle	59
TWS	true wind speed	59
USB	Universal Serial Bus	
VMG	velocity made good	97
VPP	velocity prediction program	94

List of Figures

1. Introduction

The oldest evidence of sailing are paintings dated before 5,000 BC. Most parts of the world were discovered with sailing boats and often trade was established. So the main purpose since that time was transportation. These journeys took several weeks and it is desired to abridge that time. In the nineteenth century, the domination of transportation decreased resulting from the invention of the steam-engine. (Püschl, 2012, p. 1ff)

At the same time, sailing became popular as sport. Several famous sailing regattas were held for the first time back then. Such as the "100 Guinea Cup" in 1851, which is the predecessor of today's "America's Cup". (Klasing, 2016, p. 410)

Regardless of for navigation or during racing, a sailor's desire is to gain maximal driving. This reduces the necessary amount of time for a route or gives a competitive edge over an opponent. To reach that goal, there are many variables influencing a boat speed that should be contemplated. Some are stable like the shape of the hull, payload and available sails with different sizes and cuts. Others change frequently like heel, setting of the sails or surrounding wind conditions.

The wind is the main force that drives a sailing boat and therefor focus of this thesis. The tool sailors are using to master the problem of gaining the maximal performance under current wind conditions, are polar diagrams. In that, the theoretical boat speed given a wind speed and the angle to the wind is illustrated. They are valid for a given boat with a specific set of sails. If a boat can be sailed with different combinations of sails, a polar diagram for each combination has to be created.

The current state-of-the-art method for creating polar diagrams and velocity prediction programs (VPPs) is the International Measurement System (IMS) defined by the Offshore Racing Congress (ORC). To make these predictions it depends on mathematical models of the forces, that arise when hull and sails are circulated by water or wind (*ORC VPP Documentation 2016* 2016, p. 18). For its generation more than 130 measurements of hull, appendages, propeller, stability, rig and sails (*International Measurement System IMS 2016* 2016, p.2 & 32f) are necessary. They can be bought but are rather expensive. ORC Certificates are officially valid for one year only (*ORC International Certificate* 2017).

There are situations where it is not reasonable to make all the required certified measurements and spend this money, for instance if the boat is chartered or the crew is non-professional. Also the model acts on the assumption that the underwater hull is clean and sails are new which has a big impact on the boat performance. But, every ambitious sailor wants to know these characteristic numbers of his boat. Thus, the goal is to find another method for creating reliable polar diagrams that respects the current conditions.

It turned out that the three components of a polar diagram, boat speed, wind speed and wind angle, are measured continuously on most sailing boats. It is intended to identify the boat's performance by recording these values during sailing and predict the characteristic curves out of the measured data. Using this technique sailors can have a feedback how good the currently given wind force is used compared with all similar previous situations.

In this master's thesis, a regression analysis is applied to the problem. The focus lies on the steps that are necessary to find a model of the real-world multivariate setting. During the first steps of the data analysis, the problem is divided into simpler tasks. After combining the smaller subtasks, the individual data points are weighted with its estimated influencing quality to the main unit. In the end, techniques of machine learning are used to select the best model out of a variety of generated models.

2. Fundamental Physics of Sailing

In the chapter that follows, the background how a sailing boat is able to be steered through water is presented. The engine of a sailing boat is the wind. Without, it would be a cue ball of the tides. The idea of using the wind for driving is older than 7000 years, as mentioned previously. Back then *square sails* as illustrated in Figure 2.1 were used. One could sail barely against the wind and the boat had a great leeway.

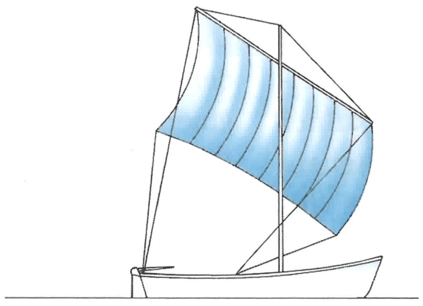


Figure 2.1.: A square sail (Klasing, 2016, p. 62).

Since the first sailing boats, the knowledge of the physical background increased and the structure of a boat and its sails changed entirely. The result are fore-and-aft sails which are available in various shapes. Compared to squared sails, they are mounted longside ship.

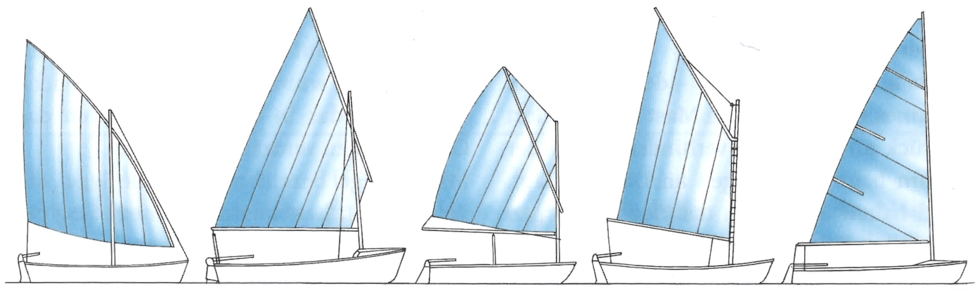


Figure 2.2.: Fore-and-aft sails (Klasing, 2016, p. 62).

2.1. The Wind

Klasing, 2016, p. 575f & p. 586ff describes wind as follows: It arises when the barometric pressure is not balanced. As a compensation, the particles in the air stream from higher

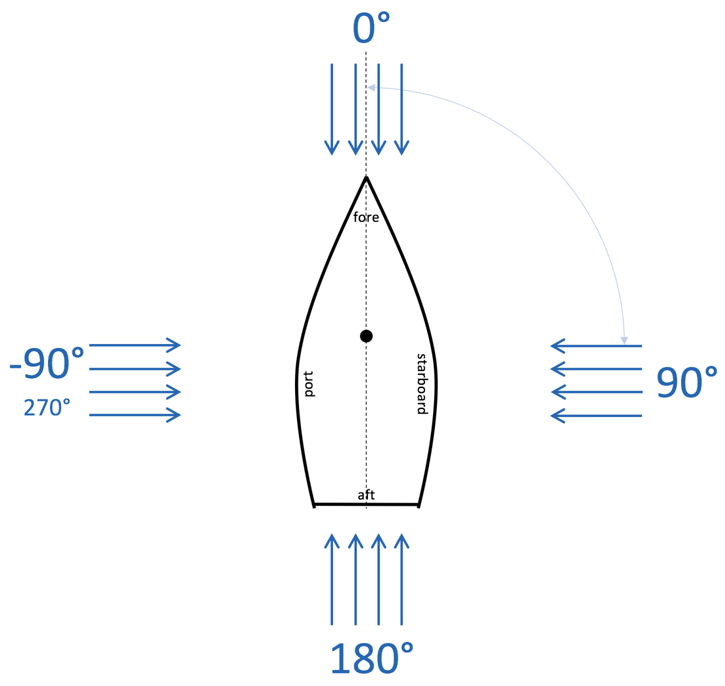


Figure 2.3.: Definition of the main wind angles relative to the boat. (selfmade)

to lower pressure. In practice, this is always the case somewhere around the globe. The most influencing factors are the sun that heats up the surroundings which in turn cause a change of the air's temperature, and secondly the rotation of the earth. The two main components of wind are the speed, that is dictated by the relative difference of pressure, and the angle.

2.1.1. Wind Angle

When viewed stationary or in weather forecast the angle of the wind is defined by the cardinal direction where it originates.

However, during sailing on a boat, the more interesting information is the relative direction to the wind. As shown in Figure 2.3 the direction is referenced to the heading of the boat. If the wind comes directly to the fore it is called zero degree angle. Wind from starboard is called 90 degrees. Wind from port is called -90 degrees or sometimes 270 degrees. The term "close-hauled" can be added after degrees to emphasize that the angle is relative to the boat direction. Why it is always important to mention if apparent or true wind is meant is described in the following subsection 2.1.2.

2.1.2. Apparent Wind vs. True Wind

On a sailing boat one has to distinguish between apparent and true wind conditions. If the observer of the wind stands still at the coast, it senses *true wind* conditions. A boatsman and also the wind gauge that is installed at top of the mast, are moving at the same speed as the boat. That means every sense and measurement includes *relative wind* due to the fact that the boat is moving. This combination of true wind and relative wind is called *apparent wind*. The connection is illustrated in Figure 2.4 and Figure 2.5.

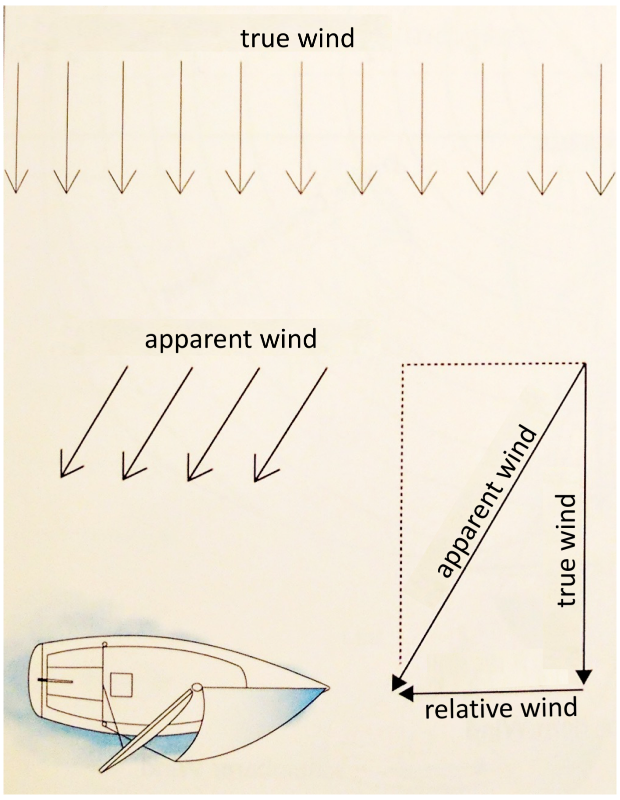


Figure 2.4.: Connection between true wind, apparent wind and relative wind (Klasing, 2016, p. 187, translated)

There are some applications, for instance navigation, which require to have a certain level of wind angle and wind speed that is independent from the boat speed (BSP). From Figure 2.4 follows that if two of the three values for relative wind, true wind and apparent wind are known, the third can be calculated by vector addition. Püschl, p. 155 defines the solution to this problem in the following two equations. The apparent wind angle (AWA) γ can be calculated from true wind speed (TWS) v_W, true wind angle (TWA) γ_w and BSP v_S as stated in Equation 2.1. BSP and relative wind have the same size, but point in opposite directions. The apparent wind speed (AWS) v can be recomposed from TWS, TWA and the previously determined AWA as defined in Equation 2.2.

$$tan\gamma = \frac{v_W sin\gamma_w}{v_W cos\gamma_w + v_S} \tag{2.1}$$

$$v = v_W \frac{sin\gamma_w}{sin\gamma} \tag{2.2}$$

2.2. Forces on a Sailing Boat

When the wind is flowing around a sailing boat it produces several forces. They are listed in Table 2.1. How they are connected during homogeneous sailing is briefly illustrated in the following section.

Table 2.1.: Main forces on a sailing boat

v_w	Speed of the true wind
γ_w	Angle between true wind and course
v	Speed of the apparent wind
γ	Angle between apparent wind and course
v_s	Boat speed
D_A	Aerodynamic drag
D_H	Hydrodynamic drag
F_A	Aerodynamic propelling force
L_A	Aerodynamic lift
L_H	Hydrodynamic lift
R_A	Total aerodynamic force
R_H	Total hydrodynamic force
S_A	Aerodynamic lateral force
α	Angle of Attack of the sail
β	Angle of drift
δ	Setting angle of the sail
ε_A	Aerodynamic glide angle
ε_H	Hydrodynamic glide angle

According to Püschl, p. 15f, a sailing boat is a coupled system of two hydrofoil-like shapes, more precisely sails and hull.

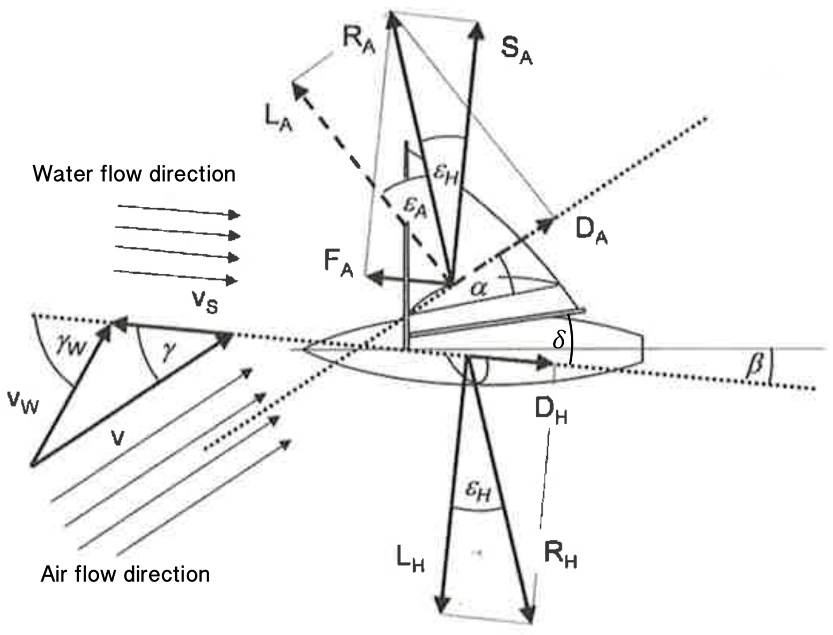


Figure 2.5.: Forces on a sailing boat during homogeneous sailing (Püschl, 2012, p. 16)

In Figure 2.5 the main forces can be seen. The relationship between true and apparent wind ($v_w, \gamma_w, v, \gamma, v_s$) is described in subsection 2.1.2.

The sail is exposed to apparent wind v with an angle of attack α. The total aerodynamic force R_A is split into an aerodynamic drag D_A and an aerodynamic lift L_A normal to it. The size of the total force is dependent on the type of sail and the angle α and can be derived from so called sail polar diagrams which are explained in detail in the following subsection 2.2.1.

The boat is moving through water with a velocity v_s thus the hull is flowed against with $-v_s$ and an angle of drift β of far less than 10 degrees (Püschl, 2012, p. 155f). The profile of the hull produces a hydrodynamic drag (D_H) in the direction of the flow and a hydrodynamic lift (L_H) normal to it. Their sum is the total hydrodynamic force R_H.

2.2.1. Equilibrium of Forces

Between the foils the first Newton's Axiom must hold which means the sum of all forces must be zero. This applies also component-by-component. Püschl concludes this requirement in the following Equations (p. 17, p. 155ff):

Equations 2.3 and 2.4 declare the same absolute value but opposite direction of the total aerodynamic and total hydrodynamic force.

$$R_A = -R_H \tag{2.3}$$

$$\gamma = \epsilon_A + \epsilon_H \tag{2.4}$$

$$S_A = -L_H \tag{2.5}$$

$$F_A = -D_H \tag{2.6}$$

The connection of the setting angle of the sail to *angle of drift*, *angle of attack of the sail* and AWA is defined in Equation 2.7

$$\gamma = \alpha + \beta + \delta \tag{2.7}$$

The following Equations 2.8, 2.9, 2.10 and 2.11 define which parameters influence the total aerodynamic (R_A) and hydrodynamic (R_H) force and its glide angles (ϵ_A, ϵ_H).

$$R_A = R_A(\alpha, v) \tag{2.8}$$

$$\epsilon_A = \epsilon_A(\alpha, v) \tag{2.9}$$

$$R_H = R_H(\beta, v_S) \tag{2.10}$$

$$\epsilon_H = \epsilon_H(\beta, v_S) \tag{2.11}$$

This shows how strongly connected hull and sails are. The total aerodynamic force R_A that is created by wind (Equation 2.8, Equation 2.9) produces the total hydrodynamic force R_H. The latter can be separated into L_H and D_H which in turn are influencing the propelling force F_A. Maximum performance can be reached only, if both foils play well together, which is explained in the following section.

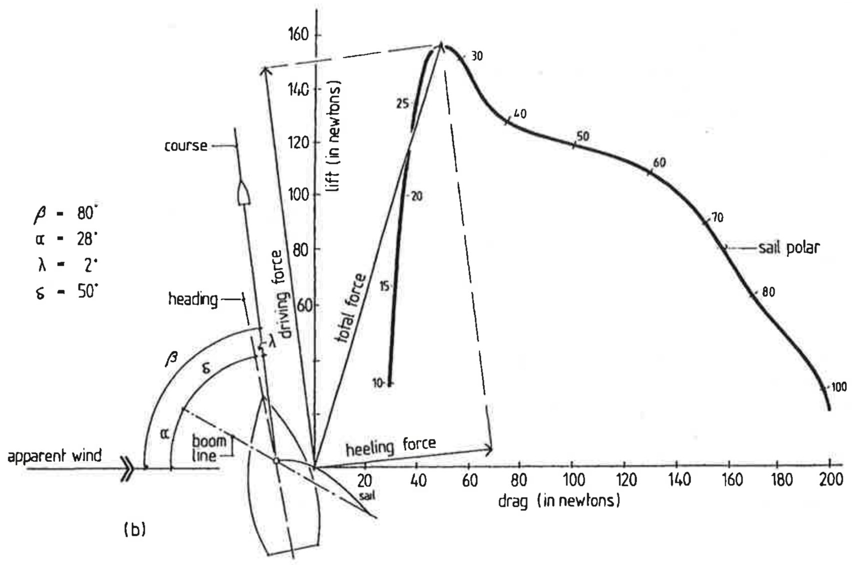


Figure 2.6.: Sail polar diagram for an AWS of about 9 knots and a sail area of 8 m^2. (Garrett, 1996, p. 62)

2.3. Use of Forces for Driving with Best Performance

The type of hoisted sails are heavily influencing the characteristics of a sailing boat. The drive a sail can create is dependent on many parameters including area and cut. As defined in Equation 2.8 for one arbitrary sail the maximal total aerodynamic force it can create is related to the angle of the wind. It is not possible to sail directly against the wind and if the wind comes directly from stern there are a lot of turbulences that will slow down the boat, so the maximum speed can be reached with lateral wind as Scheer states. In this section it is shown that the concrete value can be derived by combining sail polar diagrams with hull polar diagrams.

A sail polar diagram is a visualization of the forces of a sail depending on the wind angle. In Figure 2.6 Garrett shows one for a wind speed of about 9 knots. The forces were collected by performing measurements on a full scale boat. The oldest technique is to tether a boat to the mole with a force meter, and set the sails. The measured *aerodynamic lift* and *drag* can be added together to the *total force* and are recorded for varying *setting angles*. The total force can also be split into the *driving force* in direction of the boat and the *heeling force* normal to it. After creating a sail polar diagram it is possible to find the optimal setting angle of the sail that maximizes the aerodynamic driving force.

As mentioned previously, when the boat is moving through water, the total aerodynamic force produces the total hydrodynamic force. The reason is that the hull is a foil under

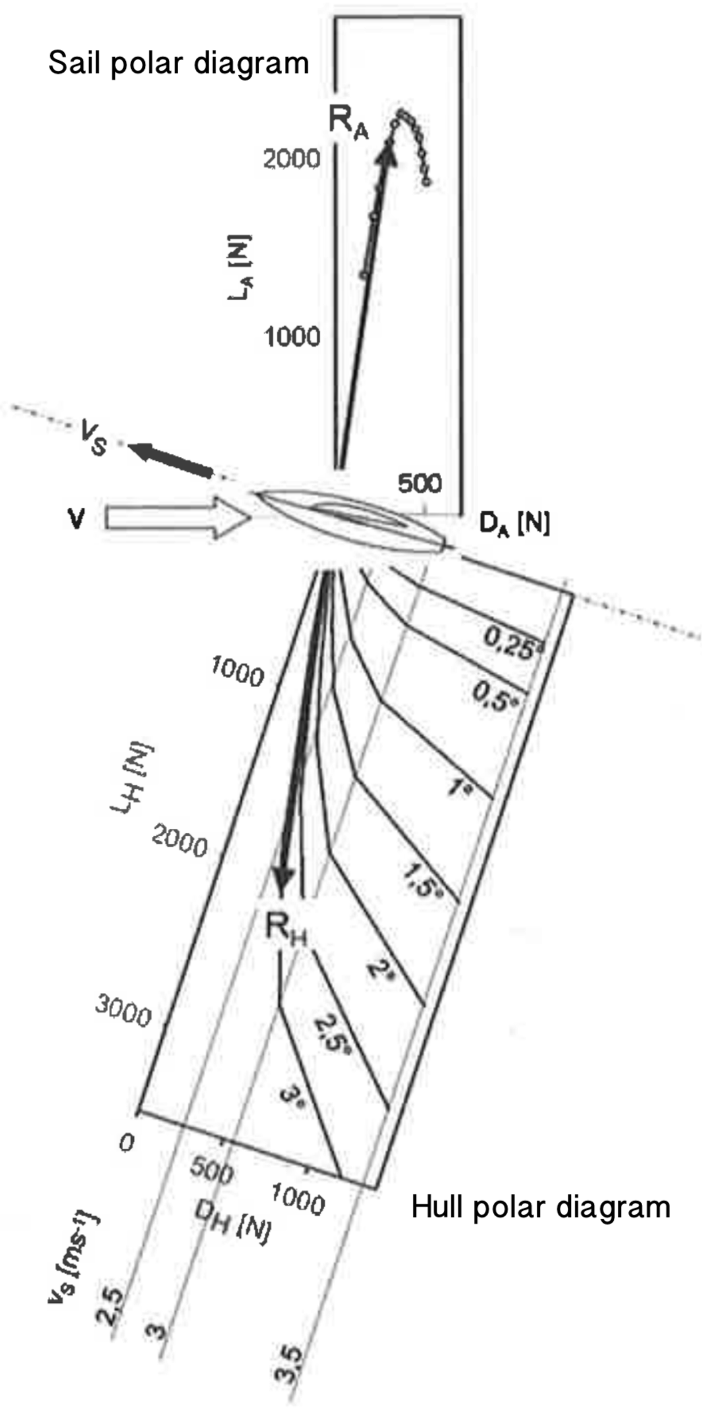


Figure 2.7.: A hull polar diagram (Püschl, 2012, p. 154).

water and it is flowed by water at an angle β. In homogeneous movements, both are of same size as described in Equation 2.3. The total hydrodynamic force can also be separated into two components which is illustrated in hull polar diagrams.

A sample polar diagram by Püschl can be found in Figure 2.7. The angles in an hull polar diagram are the hydrodynamic angle of drift β. In this figure it has a range from 0 to 3 degrees. The distribution of the total hydrodynamic force R_H to lift L_H and drag D_H in Newton can be derived.

Aside from that, the current heel influences the performance of a boat, but only in a negative way as Garrett, p. 76 describes it:

> "Whichever way one looks at it, there is a loss of performance due to heel which arises solely from the geometry of the situation."

Thus heeling can be omitted when analyzing additional forces that may increase the overall performance.

Thus far all positive influences of the wind (speed of the wind v_w, angle of the apparent wind γ) on the boat speed were discussed. In the following subsection, it is presented how those influences can be combined.

2.3.1. Performance Polar Diagram

In a performance polar diagram the BSP under some given wind conditions can be derived. When combining the information of sail polar diagram, hull polar diagram and some other parameter it is possible to predict the performance of a boat for a planned course under given wind conditions after following 7 calculation steps (Garrett, 1996, p. 69f). These steps can be calculated for several wind angles and speeds to create a performance polar diagram.

A line in a performance polar diagram is the BSP in ms^{-1} for a fixed wind speed as a function of wind angle. In Figure 2.8 such a line for a TWS of $3.5ms^{-1}$ is illustrated. The continuous line illustrates the speed for sailing with the main sail only and dashed line combines the main sail with a spinnaker sail. It can be seen for the main sail that the boat will be about 1.5 times faster at an wind angle of 75 degrees compared to an angle of 45 degrees. References to forces, like in sail- or hull polars, are not part of a performance polar diagram.

Another representation of performance polar diagrams is illustrated in Figure 2.8. On the left side the BSP for apparent wind is plotted and on the right side the relative wind

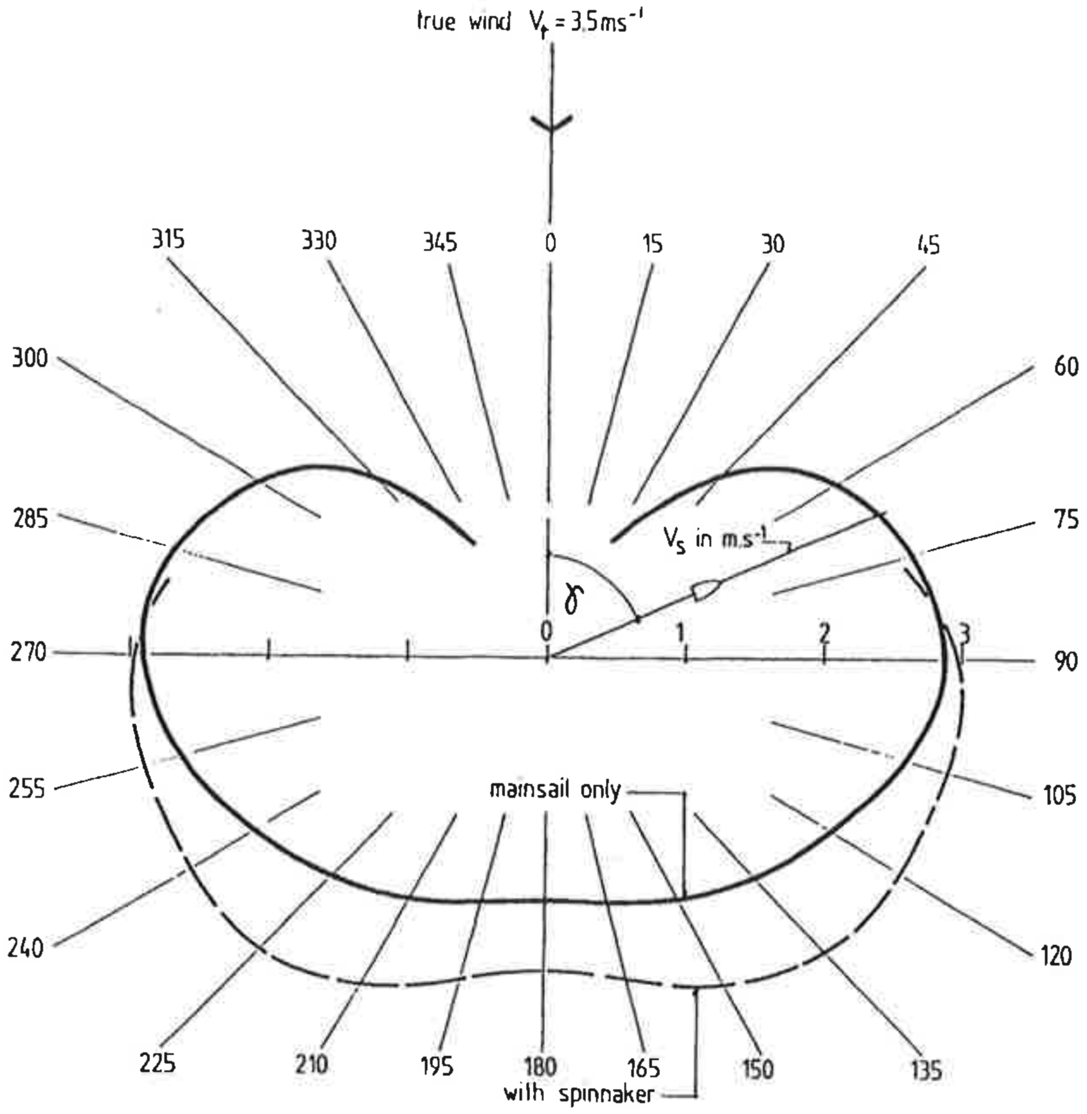


Figure 2.8.: A performance polar diagram (Garrett, 1996, p. 71).

is computationally eliminated which results in the BSP for true wind conditions. The blue line shows the characteristic of a main sail with a Jib as foresail, whereas in green, the larger, lightweight and bellied symmetric spinnaker as foresail is outlined. The latter is more than one knot faster when sailing downwind.

With a performance polar diagram of all available sails of a boat, a sailor can determine which sail to choose for desired courses and select its best angle for the highest velocity. It also is one of the main inputs of weather routing (Stelzer and Pröll, 2008). This technique is used for longer distances. Its goal is to minimize journey time by analyzing forecasts of wind and water streams.

Originally performance polar diagrams were created by tying a boat to a mole or mooring, set a sail and measure the pull on the ropes. Nowadays, area and shape of sails are measured and mathematical models give a good prediction on the possible forces. More details about this technique are provided in the following subsection.

2.3.2. Velocity Prediction Program

The most popular model is developed by the Offshore Racing Congress (ORC) and is also used to compare the performance of different ships at official regattas (*International Measurement System IMS 2016* 2016). It combines mathematical models for

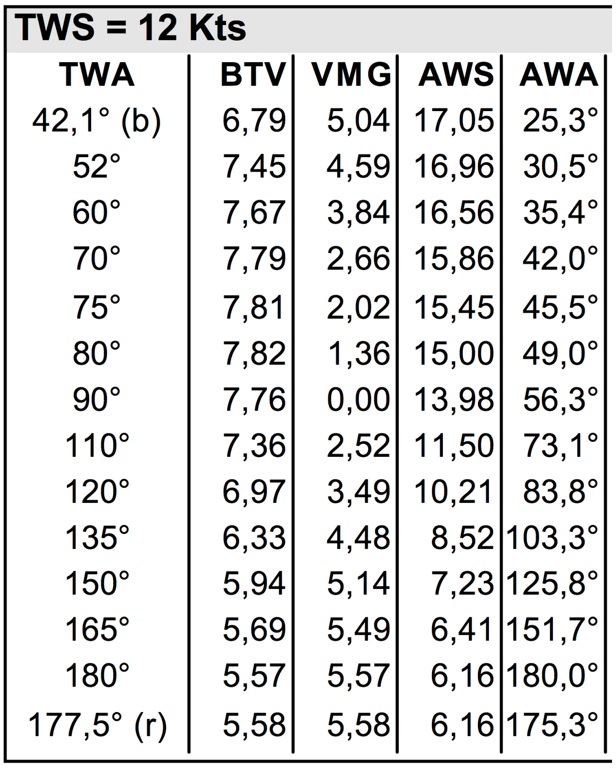
> ”the aerodynamic driving force, the heeling moment from the above water part of the hull and rig, the drag of the hull keel and rudder and the righting moment from the hull and crew” (*ORC VPP Documentation 2016* 2016)

to create a rating for each boat.

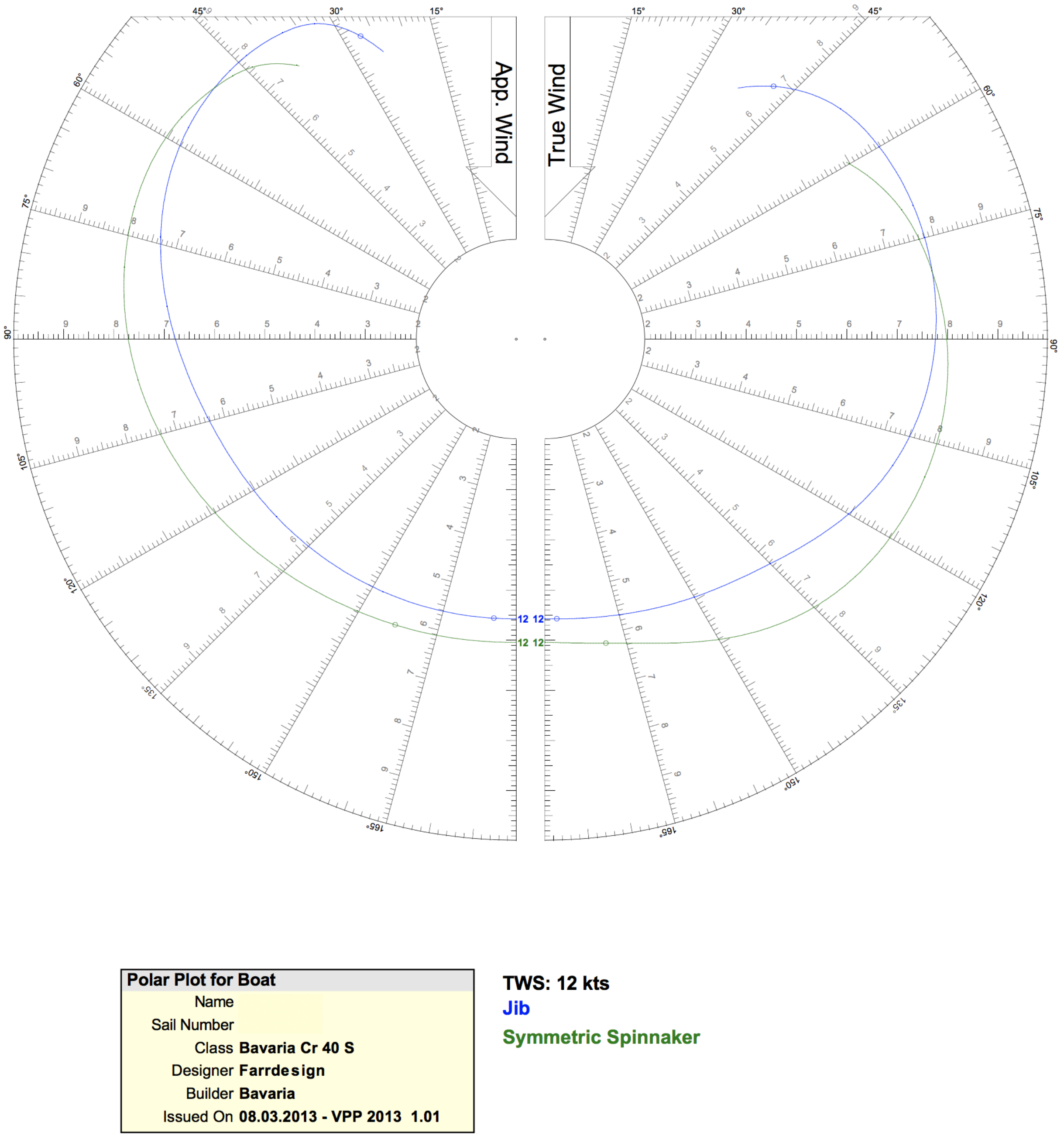
For its generation more than 130 measurements of hull, appendages, propeller, stability, rig and sails (*International Measurement System IMS 2016* 2016) are necessary. The current setting of waves is not part of the model. A new version, updated with the latest scientific experiences, is published every year.

The model calculates interpolation points Table 2.2 for wind speed of 6, 8, 10, 12, 14, 16 and 20 knots at wind angles of 52, 60, 75, 90, 110, 120, 135, 150 and 180 degrees. With these points a polar diagram like in Figure 2.9 can be interpolated. Each point on a line has the same wind speed. If a point is farer away from the center a higher boat speed at this angle is possible.

Table 2.2.: Theoretical BTV and VMG at 12 knots TWS of a Bavaria CR 40S. (*ORC Club Certificate Bavaria CR 40S* 2012)



TWS = 12 Kts				
TWA	BTV	VMG	AWS	AWA
42,1° (b)	6,79	5,04	17,05	25,3°
52°	7,45	4,59	16,96	30,5°
60°	7,67	3,84	16,56	35,4°
70°	7,79	2,66	15,86	42,0°
75°	7,81	2,02	15,45	45,5°
80°	7,82	1,36	15,00	49,0°
90°	7,76	0,00	13,98	56,3°
110°	7,36	2,52	11,50	73,1°
120°	6,97	3,49	10,21	83,8°
135°	6,33	4,48	8,52	103,3°
150°	5,94	5,14	7,23	125,8°
165°	5,69	5,49	6,41	151,7°
180°	5,57	5,57	6,16	180,0°
177,5° (r)	5,58	5,58	6,16	175,3°



Polar Plot for Boat	
Name	
Sail Number	
Class	**Bavaria Cr 40 S**
Designer	**Farrdesign**
Builder	**Bavaria**
Issued On	**08.03.2013 - VPP 2013 1.01**

TWS: 12 kts
Jib
Symmetric Spinnaker

Figure 2.9.: Certified performance polar diagram by the ORC for a TWS of 12 knots and two different sails. (*ORC Club Certificate Bavaria CR 40S* 2012)

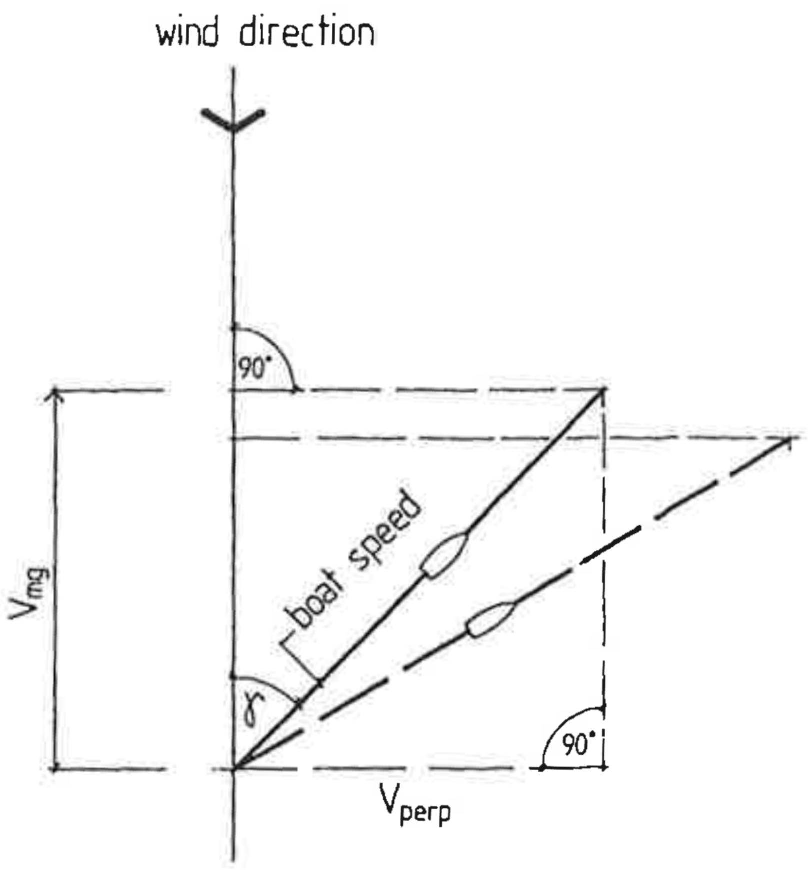


Figure 2.10.: VMG of a boat, sailing two different angles to the wind (Garrett, 1996, p. 80).

2.3.3. Top-Speed vs. Velocity Made Good

Most of the time during sailing, top-speed is not the preferred way to travel. Thats because a sailor wants to reach a specific location and often the goal is set directly against the wind, like in regattas. In Table 2.2 it can be seen that, given a true wind of 12 knots, the angle where the boat probably can cover the longest distance in an amount of time is 80 degrees close-hauled. Sailing strictly with an angle of 80 degrees would lead to a great detour. In this example, after an hour the boat would only be 1.36 nautical miles nearer to the goal. This value is called velocity made good and a sailor pursue the objective to maximize it.

Garrett depict this in Figure 2.10 where a good proportion of speed and direction windwards will result to the temporally shortest route. The VMG in direction against the wind is greater, even though the BSP of the dashed line is higher. This is traced back to the larger wind angle γ.

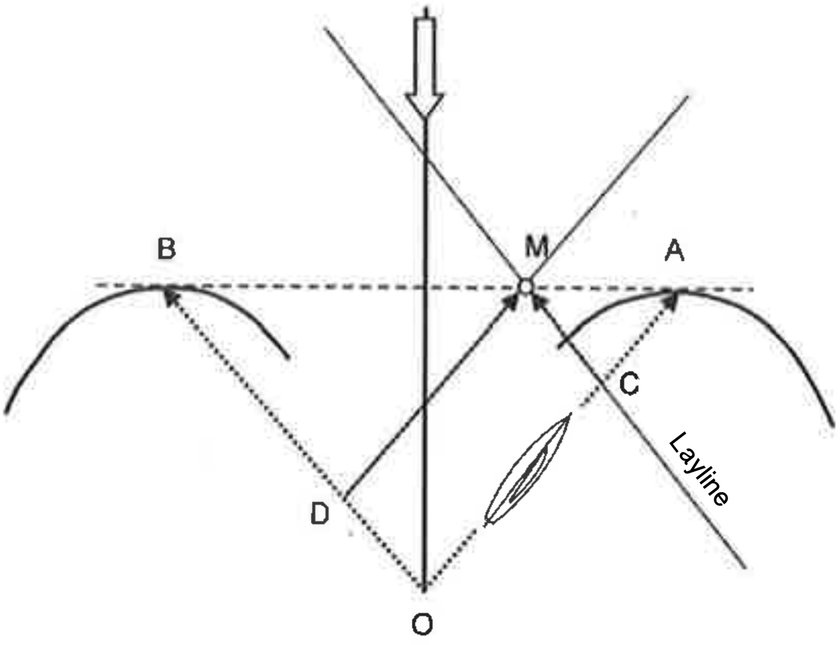


Figure 2.11.: VMG during upwind tacking (Püschl, 2012, p. 160).

In practice, every point on the convex hull around of a symmetric performance polar diagram can be reached with the shortest amount of time by adding vectors parallel to points that are both on the polar diagram and the convex hull. This is used during upwind tacking and the calculation of VMG.

In Figure 2.11 the directions with the highest VMG are $\overrightarrow{OA}$ and $\overrightarrow{OB}$. To reach point M, the fastest routes are $\overline{OC} + \overline{CM}$ or $\overline{OD} + \overline{DM}$.

The same techniques is used when the boat has multiple sails and the goal is between the top-speed of two different sails. This is illustrated in Figure 2.12 in an performance polar diagram that combines top speed of the boat regardless of the hoisted sail. The vectors $\overrightarrow{OA}$ and $\overrightarrow{OB}$ represent the top-speed of the sails. It is possible to reach goals that are farer away as the concave part in a combined polar diagram by changing the sail. The fastest route to point M is $\overline{OC}$ with the gennaker and $\overline{CM}$ without. By just using a gennaker, only point N would be reached.

The VMG is indicated in VPP with a small circle, but can derived by constructing a line normal to the wind direction. This is illustrated in Figure 2.13.

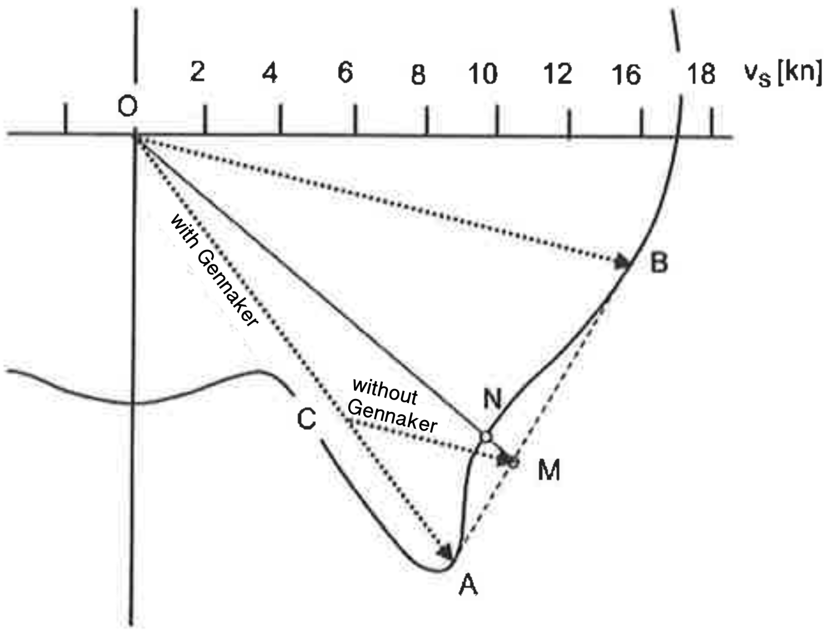


Figure 2.12.: VMG with multiple sails (Püschl, 2012, p. 161).

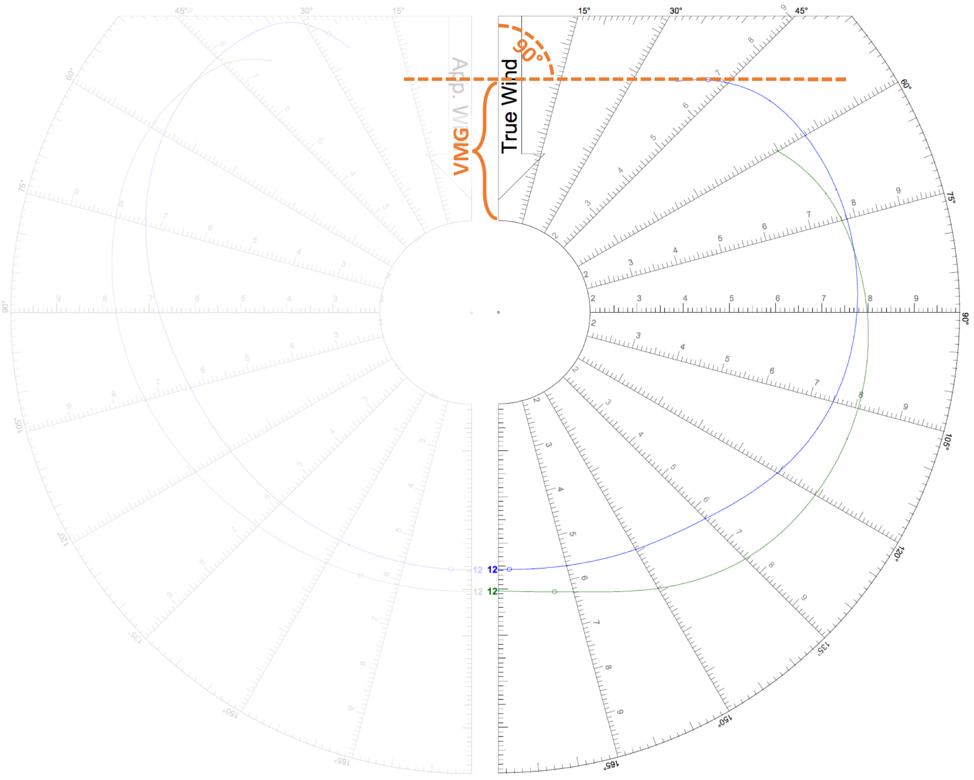


Figure 2.13.: Derive VMG in an ORC VPP (adjusted from *ORC Club Certificate Bavaria CR 40S* 2012).

2.3.4. Hull Speed

According to Klasing, p. 194 every boat has a maximal hull speed it can reach in displacement mode whereat the resistance of the hull gets nearly insuperable. This is the speed at which the stern wave spreads out directly behind the stern. The hull speed R can be expressed relative to the waterline length as defined in Equation 2.12.

$$R[kn] \approx 2.43 \times \sqrt{length_{waterline}[m]} \tag{2.12}$$

Modern, light boats can overcome this limit by producing uplift and riding on the bow wave.

3. Data Analysis

In this chapter the scientific method of *data analysis* that is used to examine the problem of the thesis is explained. Data Analysis is the umbrella term for statistical and probabilistical analysis of available or measured data with the goal to identify relationships and gather information. It is a broad field of mathematics used for research in many fields including geography, environmental science and engineering.

According to Acevedo, 2012, p. 6 data analysis should use the following general steps:

"

1. Problem definition
   a) Define the questions to be answered by the analysis
   b) Define possible assumptions that could simplify the questions
   c) Identify components and their relationships aided by graphical block diagrams and concrete maps
   d) Identify domains and scales in time and space
   e) Identify data required and sources
   f) Design experiments for data collection
   (in this case would need to go to step 4 and return to step 2)
2. Data collection and organization
   a) Measurements and experiments
   b) Collecting available data
   c) Organize data files and metadata
3. Data exploration
   a) Variables, units
   b) Independent and dependent variables
   c) Exploratory data analysis
   d) Data correlations
4. Identification of methods, hypotheses and tests
   a) Identify hypothesis
   b) Identify methods and their validity given the data

5. Analysis and interpretation
    a) Perform calculations
    b) Answer to the question that motivated the analysis
    c) Describe limits of these answers given the assumptions
    d) Next steps and new hypothesis
6. Based on results, return to one of steps 1-4 as needed and repeat

"

The iterative procedure is illustrated in Figure 3.1 Each of the step can be made with several different techniques. A variety of them are described in the following sections. Step 6 points out, that data analysis is an iterative method. The result of the previous iteration should be used to improve or extent the next hypothesis.

In the following sections, the steps of the data analysis are addressed in detail.

3.1. Problem Definition

In this step the goal of the analysis is set. A general definition of where the data comes from and what the output might be. A *3-tier architecture diagram* could help identify the separate sources of data and the processing steps to the desired output. It should especially contain the source of the data.

3.2. Data Collection and Organization

During data collection the data is aggregated. The sources are publicly available databases or generation with previously defined special experiments. If the desired measuring value can not be measured directly it is sometimes feasible to do indirectly measurements. For instance it is not possible to meter true wind speed on a moving object, but when doing two separate measurements of the apparent wind and the speed the object is moving the true wind speed can be derived (subsection 2.1.2).

An relevant stage during organization of the data is its normalization. Different variable ranges or measurement systems should be identified and resolved. Afterwards the data should be well prepared and easy accessible for the following steps of data analysis.

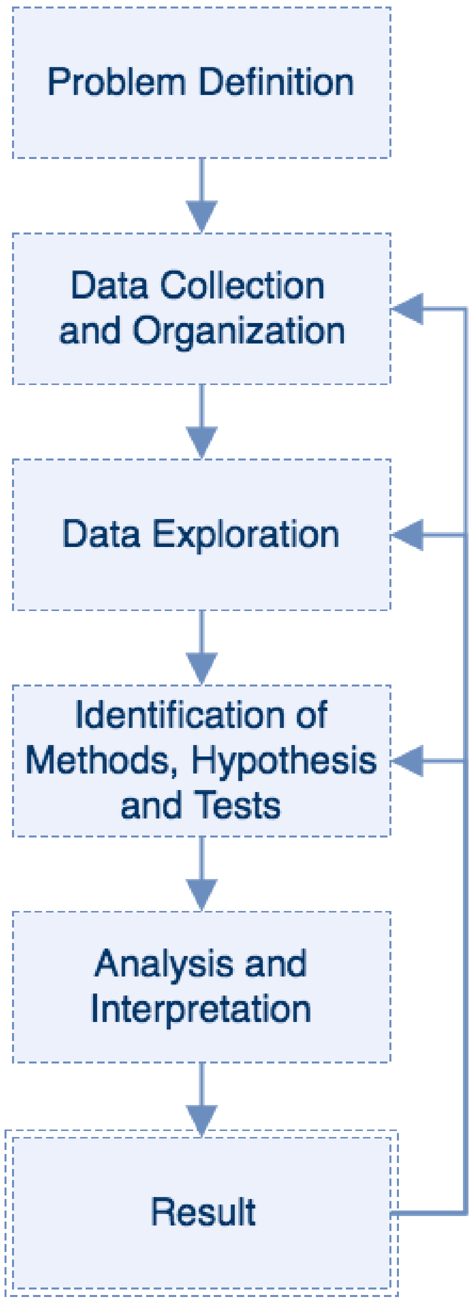

Problem Definition
Data Collection and Organization
Data Exploration
Identification of Methods, Hypothesis and Tests
Analysis and Interpretation
Result

Figure 3.1.: A state diagram of a typical data analysis according to Acevedo, 2012, p. 6 (selfmade).

3.3. Data Exploration

Step 3 mostly uses *descriptive statistic* to gain knowledge of the data. It characterizes the data without making inferences or predictions. Some visual tools are for instance histograms, boxplots or scatter plots. They are very helpful to get a first insight of the data.

Basic outlier detection can be done using interquantile range. With knowledge about the data one can ignore uninteresting samples and select a subset of the data. The circumstance of missing values should be handled with either removal of the whole entry or imputation. (Hastie, Tibshirani, and Friedman, 2009, p. 332)

3.4. Identification of Methods, Hypothesis and Tests

With the information of the exploratory data analysis of the previous step one can make a hypothesis how the available data behaves. An hypothesis is a specific question on the data and can include one or more models. A model always is a simple mapping of the reality.

When it comes to finding a solution to a hypothesis, the field of *inferential statistics* is entered. A main difference to descriptive statistic is, that inferential statistic is used under the assumption that the available data is a subset of much bigger population. The hypothesis of how this population conducts is tested for accuracy on the available data during the analysis.

There are always some influencing factors that lead to an outcome. In more technical terms, a predictor of a dependent variable of one or more independent variables should be found with this analysis and

> ”the mathematical nature or structure of the predictor determines the type of method”. (Acevedo, 2012, p. 7)

Acevedo, 2012 also states that a frequent technique for finding a predictor is *regression* which is described in the following subsection.

Table 3.1.: Sample: Dependent and independent variables (Backhaus et al., 2016, p. 64)

Question	Dependent variable	Independent variable
	Y	$X_1, X_2, ..., X_J$
Is the total revenue of a salesman dependent on the amount of customer calls?	total revenue per salesperson	amount of customer calls per salesperson per period
What is the influence of an increase of the price on the quantity of sales, if the amount for advertising is increased at the same moment?	quantity of sales	price, advertising, merchandising

3.4.1. Regression Analysis

Regression Analysis is one of the most common statistical methods. A reason for its popularity among others is its flexibility. It is used for finding the relationship of a dependent variable to one or more independent variables.

Some examples can be found in Table 3.1.

In an deterministic environment the relationship can be defined as in Equation 3.1. But in the real world the relationships have unknown influences which can be described as random noise. They are represented by the variable u in the stochastic model Equation 3.2. This is the actual equation that is solved during regression analysis. (Backhaus et al., 2016, p. 64ff)

$$Y = f(X) \tag{3.1}$$

$$Y = f(X) + u \tag{3.2}$$

Based on the data it can be chosen from a wide range of regression types amongst others:

- Parametric vs. nonparametric regression
- Linear regression vs. nonlinear regression
- Simple regression vs. multiple regression
- Spatial autoregressive vs. autoregressive time series

Objective Function equation that should be minimized by changing its parameters. The unparameterized *independent variable* is the result of one or more *dependent variables* which parameters are changed

Residual difference of one data point to the objective function

Simple regression or univariate regression describes that there is only a single variable to solve the equation for. In multiple or multivariate regression an arbitrary number of independent variables influencing a dependent variable. With a low number of independent variables it is sometimes helpful to make hypotheses for a subset of features and combine them afterwards. It is important to know that the resulting multivariate model is not automatically valid even if all univariate cases are acceptable. If the number of variables exceed a certain threshold techniques to reduce this number like *principal component analysis* can be made to identify the important features.

Parametric Regression marks that the model that should be estimated is completely described by a set of parameters. For instance a linear model as described in subsection 3.4.1. On the contrary, a nonparametric regression also has parameters, but they are unknown and depend on the data. A member is for instance Gaussian process regression. Compared to parametric regression, nonparametric regression is difficult to interpret. (Susmel, 2014, p. 3)

Robust Regression Often a small amount of data points have a great disturbance value and do not fit into the majority of values. The reasons for this observations can be manifold and these sometimes real errors are affecting the result. There are several methods that automatically identify such values and leave these out during calculation of the optimum. They are called *outlier*.

Fernandes and Leblanc illustrate the range of the result of multiple non-robust methods compared to robust methods in Figure 3.2. The four likely outliers pull the non-robust models to the left. This difference can be best recognized when comparing the non-robust *Ordinary least square SR on LAI* regression to the robust *TheilSen regression* (TS). The characteristics of these methods will not covered here in detail.

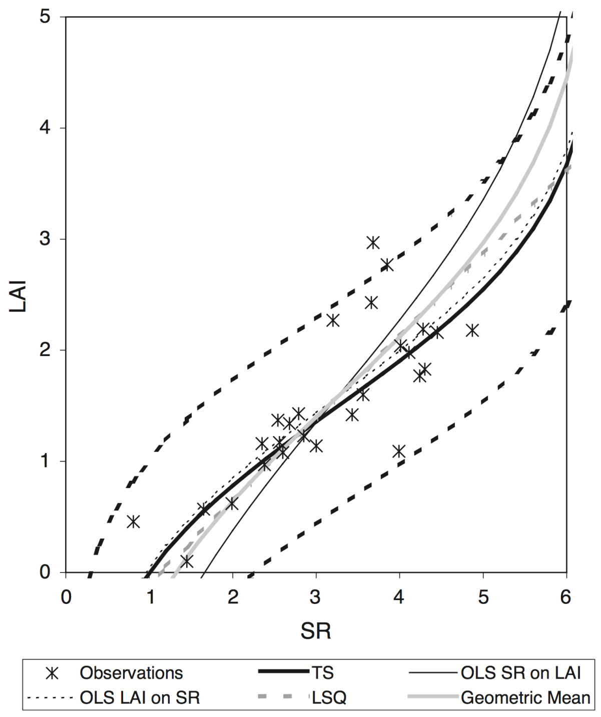


Figure 3.2.: Comparison of the result of some robust and non-robust regression methods (Fernandes and Leblanc, 2005, p. 310).

Scaling Beside a model, which is the mathematical representation of a hypothesis, regression analysis needs several complete sets of data as input. A strongness of regression analysis is that they may also include contradictory values. There are several regression methods that are sensitive to the scaling of the data. Thus the data that is used for training the model should be centered to the mean and scaled to have a variance of one. (scikit-learn-developers, 2016)

As output the analysis returns a model that was fitted to the loaded data and indicators of the quality of that model. These indicators vary from the different regression types, which are described in the following subsections.

Linear Regression

The most widespread method is the method of ordinary least squares (OLS). It is the standard method in multiple fields of application for fitting a mathematical function to a set of data points. The squared error should be minimized which means the function will be as close as possible to the measured data. The objective function for this problem has the form:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x) \longrightarrow min \tag{3.3}$$

were each r_j is referred as residual and it can be assumed that $m \geq n$ (Nocedal, 2006, p. 245f). By minimizing the sum of residuals this function will be minimized and the chosen values for the parameters of the model then will best represent the dataset. In linear case the model is defined as linear combination of non-linear basis-functions.

$$f(x) = b(x)^T c = b(x) \cdot c \tag{3.4}$$

For instance to fit a quadratic, bivariate polynomial, d = 2, m = 2 and therefore $b(x) = [1, x, y, x^2, xy, y^2]^T$ the resulting linear system of equations looks like Figure 3.3. The big

$$\sum_i \begin{bmatrix} 1 & x_i & y_i & x_i^2 & x_i y_i & y_i^2 \\ x_i & x_i^2 & x_i y_i & x_i^3 & x_i^2 y_i & x_i y_i^2 \\ y_i & x_i y_i & y_i^2 & x_i^2 y_i & x_i y_i^2 & y_i^3 \\ x_i^2 & x_i^3 & x_i^2 y_i & x_i^4 & x_i^3 y_i & x_i^2 y_i^2 \\ x_i y_i & x_i^2 y_i & x_i y_i^2 & x_i^3 y_i & x_i^2 y_i^2 & x_i y_i^3 \\ y_i^2 & x_i y_i^2 & y_i^3 & x_i^2 y_i^2 & x_i y_i^3 & y_i^4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \sum_i \begin{bmatrix} 1 \\ x_i \\ y_i \\ x_i^2 \\ x_i y_i \\ y_i^2 \end{bmatrix} f_i.$$

Figure 3.3.: OLS: sample system of equations (Nealen, 2004, p. 1)

advantage is, given these preconditions, there exists an analytical solution. An existing

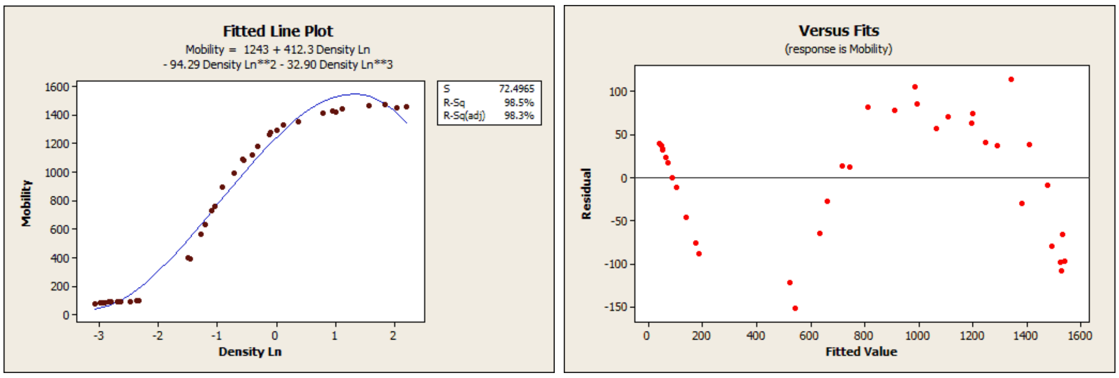


Figure 3.4.: Fitted line and residual plot of a linear model to nonlinear data (Frost, 2011).

minimum can be found in one step and it is not necessary to do an iterative linear approximation.

The main weakness is, that this method is not robust against outlier. It also has some limitations and preconditions the resulting model should be validated against. These are described in subsection 3.4.2.

Nonlinear Regression

Sometimes linear Regression is not flexible enough to describe the world's phenomenons. Then, the more complex nonlinear models can increase the quality of the analysis.

As an example in Figure 3.4, a linear fit of a data set with relatively good R^2 is illustrated. However the residual plot on the right shows a pattern which is an indicator for a bad fit.

When fitting a nonlinear model to the same data set as in Figure 3.5, the residual plot shows the wanted randomness.

Due to the lack of an analytical solution the optimum must be with iterative estimations. There is no guarantee that this algorithm converges and the global minimum is found. Furthermore, quality of the result is frequently influenced by the starting point.

Nonlinear models are nonlinear in its variables and parameters. This difference is illustrated in Figure 3.6. The optimal value for parameter γ is part of the analysis and prevents a conversion into a linear model.

As mentioned previously the solution to nonlinear models has to be found in iterations.

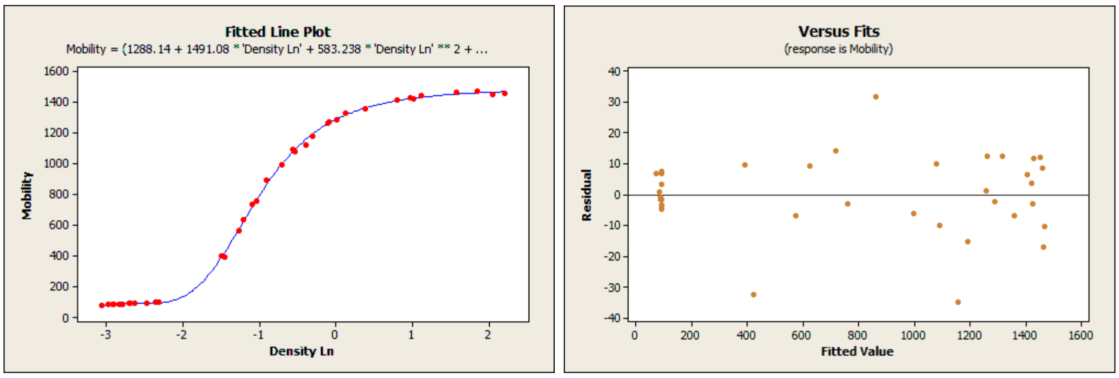


Figure 3.5.: Fitted line and residual plot of a nonlinear model to nonlinear data (Frost, 2011)

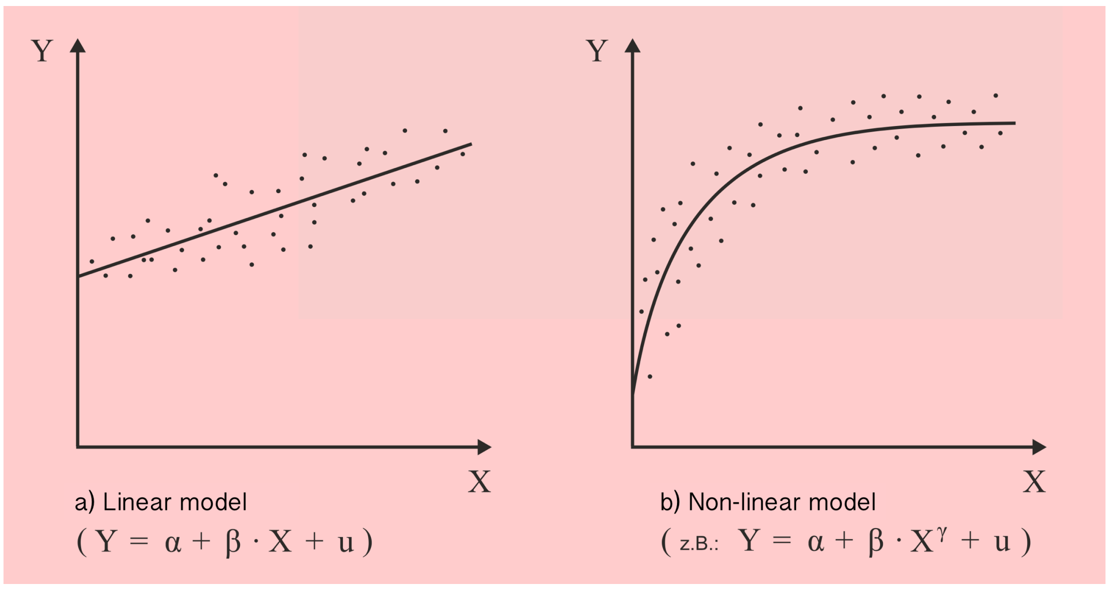


Figure 3.6.: Comparison of linear and nonlinear models (Backhaus et al., 2016, p. 574).

Gauss-Newton Method is the most simple algorithm and is an adaption of the Newton Method followed by a line search. It is more efficient by replacing the computationally expensive calculation of the Hessian matrix with an approximation. The standard Newton equations are defined in 3.5.

$$\nabla^2 f(x_k) p = -\nabla f(x_k) \tag{3.5}$$

The gradient $\nabla f(x_k)$ and the Hessian matrix ∇^2 can be transformed to include the Jacobian matrix as explained in Equation 3.6 and Equation 3.7 (Nocedal, 2006, p. 246f).

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x) \nabla r_j(x) = J(x)^T r(x) \tag{3.6}$$

$$\begin{aligned} \nabla^2 f(x) &= \sum_{j=1}^{m} \nabla r_j(x) \nabla r_j(x)^T + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) \\ &= J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) \end{aligned} \tag{3.7}$$

At the solution the term $J_k^T J_k$ is significantly bigger as the rest of Equation 3.7, thus it is a good approximation for the individual residual Hessian matrix ∇^2. Given the resulting Equation 3.8 for Gauss-Newton, in one iteration the calculation of the Jacobian matrix is the most expensive part to retrieve the next search direction p_k. It is a valid descend direction for a line search under the condition that the Jacobian J_k has full rank and the gradient ∇f_k is nonzero.

$$J_k^T J_k p_k^{GN} = -J_k^T r_k \tag{3.8}$$

A newton step is then made in this direction. This is repeated until the algorithm converges (Nocedal, 2006, p. 254ff).

Levenberg-Marquardt Method uses the same approximation for the Hessian matrix as *Gauss-Newton* but is calculating the following step with a *trust-region* strategy instead of a line search. This brings major advantages if the Jacobian matrix is not full-rank.

The main difference to the line search method is that size and direction of the trust region step are calculated at the same time. The new solution has to be within the trust region defined (in 2 dimensions) by a circle with the radius Δ.

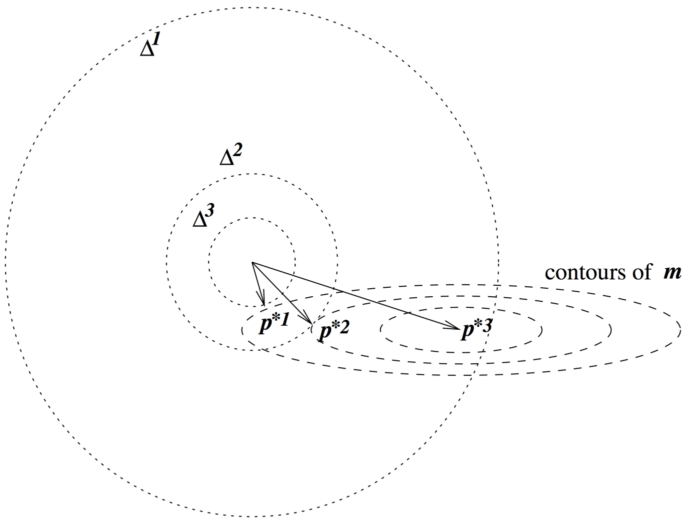


Figure 3.7.: Trust region: iteration step of 3 various step-sizes with different length and direction (Nocedal, 2006, p. 70).

For example: In Figure 3.7 the minimum of a quadratic function m is wanted. With various step sizes Δ^1, Δ^2 and Δ^3 the Newton step lead to solutions p^{*3}, p^{*2} and p^{*1} with diverging length and direction.

To find the solution of one iteration in three dimensions Equation 3.9 has to be solved, according to Nocedal, 2006, p. 258.

$$\min_p \frac{1}{2}\|J_k p + r_k\|^2 \qquad with \|p\| \leq \Delta_k \tag{3.9}$$

For the next iteration the trust region radius is adjusted. If the solution lies outside the trust region ($p > \Delta$) the step is not made but the region is increased. The size remains the same if the solution is at the border of the area ($p = 1$) and Δ is decreased if the best solution is strictly inside the trust region.

The Levenberg-Marquardt algorithm terminates if the relative change in an iteration becomes lower than a threshold ϵ.

Weighted Regression

According to National Institute of Standards and Technology, 2013 standard linear and nonlinear least squares regression require the precondition that

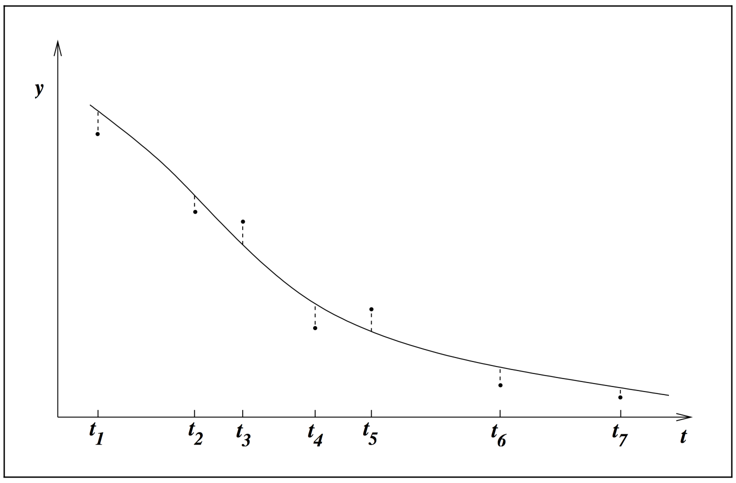


Figure 3.8.: Standard curve fit through some measured data at points in time t (Nocedal, 2006, p. 248).

> "each data point provides equally precise information about the deterministic part of the total process information".

With prior knowledge about the data one can assume that some data points are more reliable than others. That means that the information that some data points have an higher error term than others can improve the quality and accuracy of the result. The objective function 3.3 is extended with an additional parameter w_j in Equation 3.10 (Shalizi, 2009, p. 1)

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} w_j \times r_j^2(x) \longrightarrow min \tag{3.10}$$

Orthogonal Distance Regression

The classical types of regression all act on the assumption that the independent variable has no or tiny measurement errors compared to the dependent variables. Pictured in Figure 3.8 the distance from the final curve to the data points that is minimized is indicated with dotted lines. In standard regression methods they are parallel to the y-axis.

Given the example of Figure 3.8 it may be the case that the exact instant of time is not exactly known. Then there are uncertainties in both axis and the additional errors have to be taken in account. The distance to the curve should then be calculated with *geometric* shortest distance, which is illustrated in Figure 3.9. This type of regression is called orthogonal distance regression.

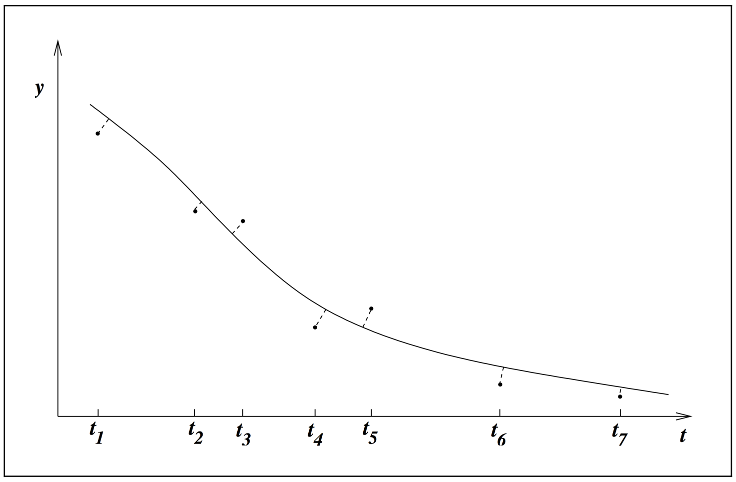


Figure 3.9.: Curve fitted with ODR through some measured data (Nocedal, 2006, p. 266).

The deviation of the result of multiple calculation methods is illustrated in Figure 3.10.

3.4.2. Hypothesis Validation

When the optimal objective function was found one has to check if the outcome is valid. There are two steps to analyze the result.

Validate Regression Function

In this subsection a definition of a number that can quantify the quality of the resulting model is gathered.

Goodness of Fit The most interesting part is the goodness of fit. That means how well does the model describe the depended variable. Backhaus et al., 2016 defines it as follows (p. 82ff): The *sum of squared residuals* (SSR, Equation 3.11) can compare two models on the same data set. A smaller SSR is a sign for a better fit, but it should be investigated together with the *coefficient of determination* (Equation 3.15).

$$SSR = \sum_{k=1}^{K} (y_k - \hat{y}_k)^2 = \sum_{k=1}^{K} e_k^2 \tag{3.11}$$

$$SSE = \sum_{k=1}^{K} (\hat{y}_k - \bar{y})^2 \tag{3.12}$$

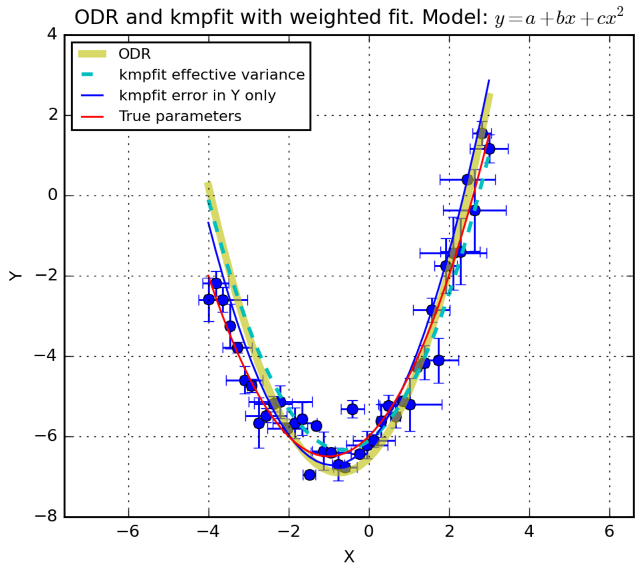


Figure 3.10.: Comparison of ODR and regression with errors in Y only. (Kapteyn Astronomical Institute, 2015)

A deviation can be separated into an explained part and a residuum. For linear models the last term in Equation 3.13 is 0, thus the *total sum of squares* (SST) equals the *explained sum of squares* (SSE) plus the SSR, compare Equation 3.14.

$$\begin{aligned} \underbrace{\sum_{k=1}^{K}(y_k-\bar{y})^2}_{SST} = \underbrace{\sum_{k=1}^{K}(\hat{y}_k-\bar{y})^2}_{SSE} + \underbrace{\sum_{k=1}^{K}(y_k-\hat{y}_k)^2}_{SSR} + \\ + \underbrace{\sum_{k=1}^{K} 2*(\hat{y}_k-\bar{y})(y_k-\hat{y}_k)}_{=0} \end{aligned} \tag{3.13}$$

$$SST = SSE + SSR \tag{3.14}$$

Coefficient of Determination - R squared Another key value is the *coefficient of determination* or *R squared*. It is defined as explained error divided by the total error (Equation 3.15) which has a range from 0 to 1. The closer *R squared* is to 1, the better the measured data are described by the model.

$$R^2 = \frac{SSE}{SST} \tag{3.15}$$

As discussed previously the third term in Equation 3.13 is 0 only during linear regression. When it comes to nonlinear regression the calculation of R^2, that relies on Equation 3.13, is not valid anymore and an alternative is demanded.

χ^2 - chi-squared The χ^2 goodness of fit test ought to be valid on nonlinear models. It is like SST an quality parameter that indicates how well the fitted model describes the measured data. The idea behind it is, that if the sample size is large, it should spread around the fitted model with a χ^2-distribution.

Barron, 1997 defines it by the following Equation 3.16.

$$\chi^2 = \sum \frac{(observed - expected)^2}{expected} \tag{3.16}$$

The goodness of fit is better if χ^2 is lower. Thus, it can be used as an objective function of a regression analysis.

Similar to the R^2 there exist a so called *reduced chi squared* χ^2_{red}. It is defined by Andrae, Schulze-Hartung, and Melchior, 2010, p. 1 in Equation 3.17.

$$\chi^2_{red} = \frac{\chi^2}{K} \tag{3.17}$$

The optimal goodness of fit would be a χ^2_{red} of 1. A number below this indicates overfitting and a number above lead that the model cannot express the variety of the data. If χ^2_{red} is high above 1, the fit is poor.

In linear case, the *degree of freedom* K is often trivially specified as in Equation 3.18

$$K = N - P \tag{3.18}$$

where N is the number of data points and P the number of parameters. He shows that this an imprecise approximation and the true value is somewhere in the range between $K = N - P$ and $K = N - 1$.

However, when it comes to nonlinear models, Andrae, Schulze-Hartung, and Melchior, 2010, p. 1 proves K can not be approximated. That lead to the conclusion that χ^2_{red} must not be used for nonlinear models. The author pointed out that there are no barriers to do χ^2-regression on nonlinear data.

Validate Preconditions

After successful validation of the regression function it is important to check the preconditions of the regression techniques. According to Backhaus et al., 2016, p. 97ff it is not allowed to perform linear regression without testing against the assumptions on the model.

The most important are:

- model is correctly specified
- disturbance values have a constant variance (homoscedasticity)
- disturbance values are uncorrelated (auto-correlation)
- disturbance values are normal distributed

which are explained in this section.

Residual plot A residual is the difference of the observed data y to the fitted function value $\hat{y}$ (Equation 3.19).

$$r = y - \hat{y} \tag{3.19}$$

A residual plot combines all residuals in a plot. It can be used to visually identify invalid preconditions, which is described in the following chapters, or modelling errors. If there is an visible pattern in the residuals it is a strong indication that the model does not fit to the data. A residual plot should show randomness.

An example residual plot for an invalid model is illustrated in Figure 3.11. The total sum of squares may be low but it is obvious that the residuals have the shape of an parabola. This leads to the conclusion that the linear model does not represent the data well.

Homoscedasticity The data should be homoscedastic which means that the variance of the residuals should be constant. To identify heteroscedasticity one can make a plot of residuals that will show a shape of a triangle like in Figure 3.12. A mathematical approach would be the Goldfeld/Quandt-Test or the method of Glesjer.

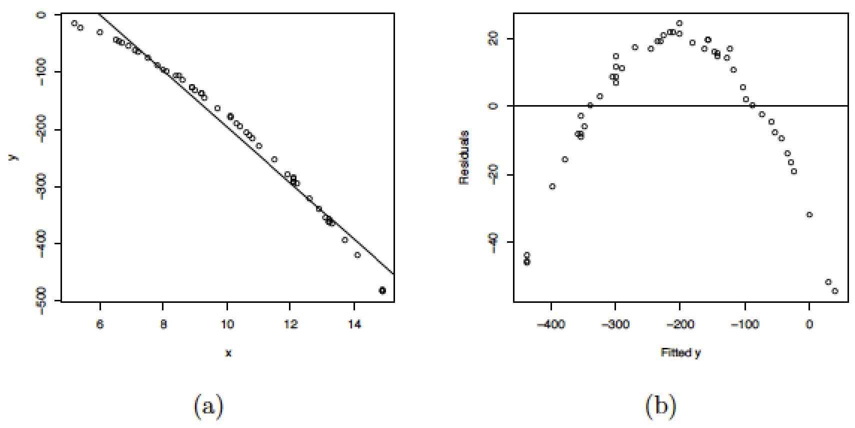


Figure 3.11.: A plot of residuals (b) of the scatter plot with a fitted straight line (a). (Department of Statistics Online Programs, 2017)

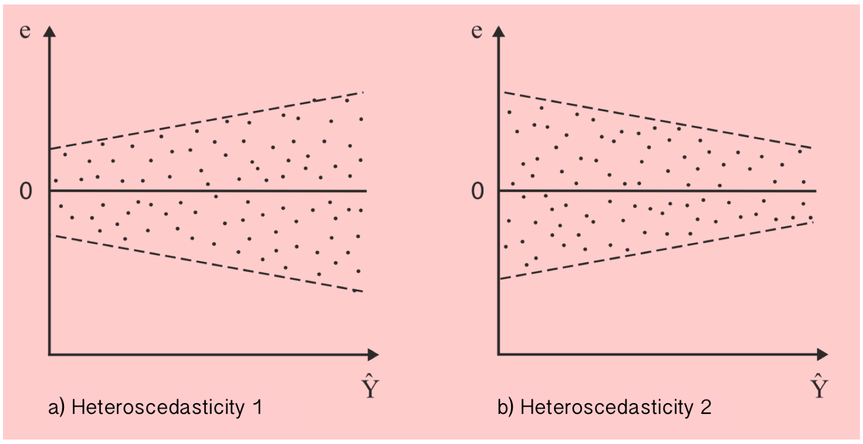


Figure 3.12.: Types of heteroscedasticity (Backhaus et al., 2016, p. 103).

Autocorrelation Another precondition is the nonexistence of autocorrelation. That means the order of the observations affect the results and therefor the residuals. This is often a phenomenon when analyzing time series. With the Durbin/Watson-Statistic it is possible to test for such an existence. The resulting value is between 0 and 4. 0 means there is a positive autocorrelation and a value of 4 that there is a negative autocorrelation. If the value is 2, there is no autocorrelation.

3.4.3. Hypothesis Selection

Guerzhoy, 2015, p. 4ff and Sarle, 2002 deal with the problem in supervised machine learning to decide what is the best classifier of a data set. The solution is to separate the data into distinct data sets called the *training set* and the *test set*.

The training set is used that the classifier learns from it, which means minimizes an arbitrary error function. Afterwards the quality of a completely trained classifier is calculated by determining the same error function with previously unknown data, the test set. The test set must not be used to adjust any parameters of the classifier. A third distinct *validation set* can be used for neuronal networks to tweak some parameters.

Because of the similarity between most neuronal networks and statistical inference, the same concept can be applied in data analysis to select the best hypothesis (Sarle, 2002). Feasible error function would be the sum of squared error or χ^2 if they are applied on the same data set.

As emphasized in (scikit-learn-developers, 2016), if the data is scaled prior to fitting, the scaling factors should be determined from the training set only. The test set is then scaled with the same scaling factors as the training set.

Over-fitting is the fact, that a model is overly complex and it just learned by heart. One has to check every model that this is not the case, because it violates the precondition of inferential statistics that the available data is a part of a larger data set. An indicator is a small training error but an high test error.

Thus the preferred hypothesis is the one with the least test error, because it best predicts new, unknown data.

3.5. Analysis and Interpretation

In the fifth step the chosen method of the previous step is applied to the hypothesis. The quality parameters should be analyzed and checked for validity. Then the result should be applied to the original problem of the analysis. It maybe can be connected with the previous iterations to a solution. In this case limitations of the result should be discussed or otherwise the following steps and hypothesis should be declared.

3.6. Iteration

After finishing an iteration, a review of the result of this and the previous iterations is made. One can finish the analysis or decides to make another iteration starting at step 1 to 4.

4. Hardware and Software

The next chapter gives a practical view of the realization of this thesis. In the first part the devices for the measurements and data collection is described. The second section delivers an insight to the developed software and design principles.

4.1. Measurement Devices

In the following section the common measurement devices on a sailing boat are described briefly.

4.1.1. Wind Gauge

Wind gauges can measure the force and direction of the wind. As explained in subsection 2.1.2 they are moving at the same speed as the boat thus are measuring apparent wind speed and apparent wind angle. They are mounted on the top of the mast and commonly consist of a wind vane and rotating wind cups.

When a boat breasts the waves, the wind gauge on the top of the mast is even more exposed to the jerky movements. That is why most manufacturers include a method of smoothing into the display unit.

Especially the wind angle has to be properly calibrated, otherwise the effect that a bow is *better* than the other would occur.

4.1.2. Speed Log

Speed logs are mounted at a position in the hull that is always below the surface of water. They are used to measure the speed of the boat relative to water. Since they tend to get dirty, they should be cleaned regularly.

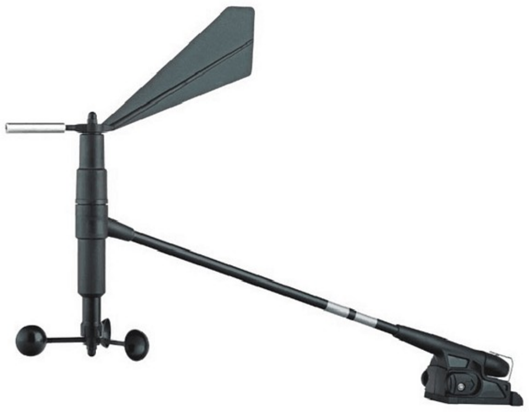


Figure 4.1.: B&G 608 Wind Sensor (Hodges Marine Electronics, 2017)

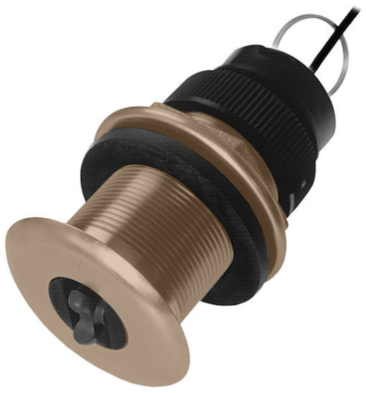


Figure 4.2.: Furuno ST-02MSB Thru-Hull Speed and Temperature Sensor (S2ware, 2017).

The output of the impeller should be calibrated by sailing a short known distance multiple times in different directions.

A speed log by the company Furuno is illustrated in Figure 4.2. It has a diameter of about 5 centimeters.

4.1.3. Global Positioning System (GPS)

Global Positioning System (GPS) is used to determine the position on a boat and the current time. A number of satellites send their position and a time-stamp to the earth. The receiver can calculate its distance to the satellite by the transit time and combine the distances to multiple satellites to a explicit position on the earth's surface. The accuracy normally is around 10 meters but can be improved with Differential GPS (DGPS) to 1-3 meters. By combining multiple positions one can also measure the absolute boat speed. (Garmin, 2017)

4.2. Data Transmission on a Boat with NMEA

The most common standard on sailing boats is by the National Marine Electronics Association. It is available in the two different types NMEA 0183 and NMEA 2000.

NMEA 0183 was, according to Betke, 2001, p. 1ff, released in 1983 and defines an electrical interface using 2 wires between a single *talker* and several *listeners*. The talker sends with a rate of 4800 baud. The specification defines a general format for the communication which is called *sentence*. Its definition can be found in Table 4.1. The talker identifier specifies the type of device like '*Velocity Sensor, Speed Log, Water, Mechanical*' (`VW`) or '*Global Positioning System*' (`GP`) whereas the sentence identifier describes the type of measurement. Thus the same measurement category can be delivered by multiple devices.

Measuring instruments deliver their current gauging every second. In Figure 4.3, Figure 4.4 and Figure 4.5 some common sentence formats are specified.

NMEA 2000 is the successor of NMEA 0183 and brings big enhancements to the transmission of NMEA sentences. The standard removes the central controller and introduces a self-configuring network with multiple transmitters and receivers.

Table 4.1.: General format of a NMEA sentence (Betke, 2001, p. 2)

`$ttsss,d1,d2,....*hh<CR><LF>`	
tt	talker identifier
sss	sentence identifier
d1,d2,....	data fields; comma-separated
hh	checksum
`<CR><LF>`	termination (carriage return + line feed)

```
                                                            12
       1         2 3       4 5        6 7   8   9    10  11|
       |         | |       | |        | |   |   |    |   | |
$--RMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,xxxx,x.x,a*hh

 1) Time (UTC)
 2) Status, V = Navigation receiver warning
 3) Latitude
 4) N or S
 5) Longitude
 6) E or W
 7) Speed over ground, knots
 8) Track made good, degrees true
 9) Date, ddmmyy
10) Magnetic Variation, degrees
11) E or W
12) Checksum
```

Figure 4.3.: NMEA sentence for *Recommended Minimum Navigation Information* (Betke, 2001, p. 14)

```
       1   2 3   4 5   6 7   8 9
       |   | |   | |   | |   | |
$--VHW,x.x,T,x.x,M,x.x,N,x.x,K*hh

1) Degress True
2) T = True
3) Degrees Magnetic
4) M = Magnetic
5) Knots (speed of vessel relative to the water)
6) N = Knots
7) Kilometers (speed of vessel relative to the water)
8) K = Kilometres
9) Checksum
```

Figure 4.4.: NMEA sentence for *Water Speed and Heading* (Betke, 2001, p. 17)

```
        1   2 3   4 5
        |   | |   | |
$--MWV,x.x,a,x.x,a*hh

1) Wind Angle, 0 to 360 degrees
2) Reference, R = Relative, T = True
3) Wind Speed
4) Wind Speed Units, K/M/N
5) Status, A = Data Valid
6) Checksum
```

Figure 4.5.: NMEA sentence for *Wind Speed and Angle* (Betke, 2001, p. 13)

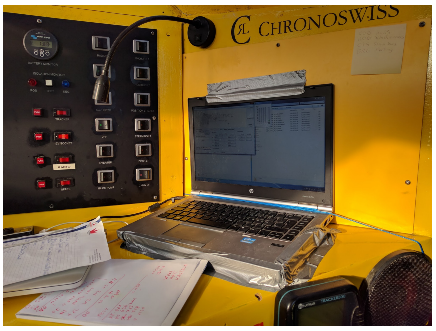


Figure 4.6.: Mounted computer to record data (selfmade).

4.3. Receiving Boat Measurements on a Computer

To receive the data of NMEA on a computer it can be connected to the serial port (or COM-port) of the computer. Due to the lack of this port on most of today's computer a virtual port over USB is used.

NMEA 0183 allows only one talker on a wire. Thus, a setup like the schema in Figure 4.7 is not possible. To receive data from more than one measurement device at the same time, a multiplexer is necessary. It merges the sentences from multiple inputs to one or more outputs. One output can then be connected with the computer, which is illustrated in Figure 4.8.

A product that combines multiplexer and virtual COM-port is ShipModule MiniPlex-2USB (*ShipModule MiniPlex-2USB* 2015).

Each of the measuring instruments deliver their current gauging every second as NMEA-sentence. To read the data on the COM port a logging software is needed. This is possible with the free configuration tool *MPX-Config2 for MiniPlex-2* 2015 or more advanced products like *Expedition 10* 2017. Those programs do have different output

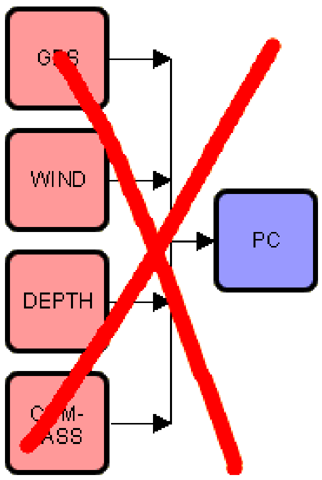


Figure 4.7.: Schema of an invalid NMEA 0183 wiring (*NMEA 0183 and Multiplexers* 2017).

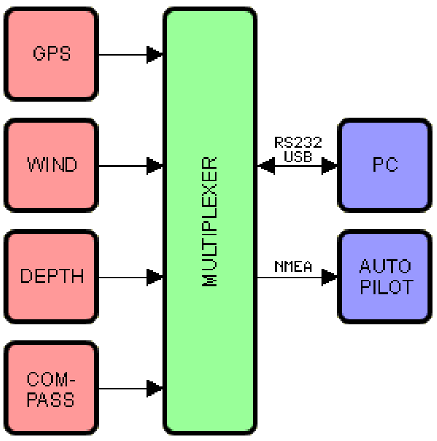


Figure 4.8.: Schema of a valid NMEA 0183 wiring with a multiplexer (*NMEA 0183 and Multiplexers* 2017)

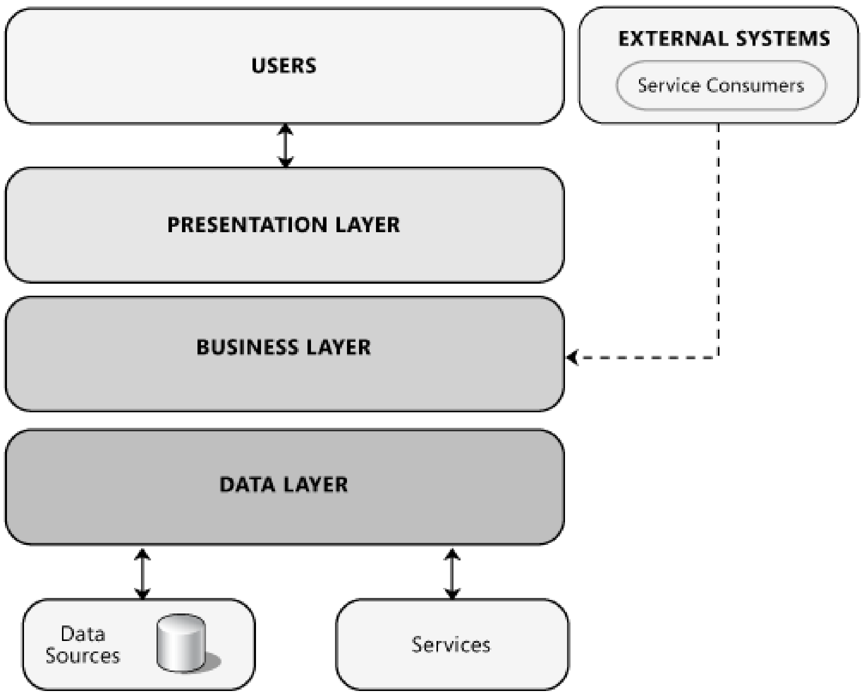


Figure 4.9.: 3-tier architecture in software development (Microsoft Cooperation, 2009, p. 56).

formats. MPXConfig2 was used for the measurements and from there it can be exported to files as a list of NMEA-sentences.

4.4. Software

In this section the background on the software is discussed that was developed during the thesis to accomplish the research question. It was taken care that the code takes advantage of use of the 3-tier architecture. This principle is illustrated in Figure 4.9.

The *data layer* is responsible for the keeping of data and has a specified access protocol. In the *application layer*, the business logic of the software is situated. It is responsible for accessing the data layer, making the necessary calculations and preparing the content for the *presentation layer*. This layer only visualizes the results for the user and delivers inputs to the application layer. With well-defined interfaces between the layers it is possible to develop them independently and that following make it easy to refactor and change the components without influencing another. (Microsoft Cooperation, 2009, p. 56f)

4.4.1. Software-Environment

For the analysis an implementation with Python 3 was chosen. Python is an

> ”interpreted, interactive, objective-oriented programming language” (*General Information* 2017)

, currently available in version 3.6.1. Version 3.0 of Python was released in the year 2008 and should be preferred to version 2.x (*Python 2 or Python 3* 2017). The analysis was performed on a MacBook Pro 2016, 15-inch running macOS Sierra. It has a built-in 76.0-watt-hour battery which lasts up to 10 hours (Inc., 2017). Python was used in version 3.6.0.

Python’s advantage is the wide choice of open source libraries for data analysis. There are many libraries covering the challenges of this analysis such as PyNMEA, SciPy and Matplotlib (subsection 4.4.4).

In the next sections the software and its usage is presented.

4.4.2. Implementation

In this subsection the particular features of the implementation are described. An overview of how the 3-tier architecture was applied, is illustrated in Figure 4.10. They can be triggered in user-defined order with a command line interface (CLI). A CLI as illustrated in Figure 4.11 is a link to the running program to dictate its behavior by entering statements. In comparison to a graphical user interface it requires less resources and it is simple to execute as series of commands with it. This offers great flexibility without high complexity in the creation of the software.

Data separation

During the analysis data from different boats were analyzed. This lead to a mechanism to separate source datasets. All input files of one boat need to be together in a folder which is called a *workspace*. This takes the advantage that all working files and outputs are made to that workspace.

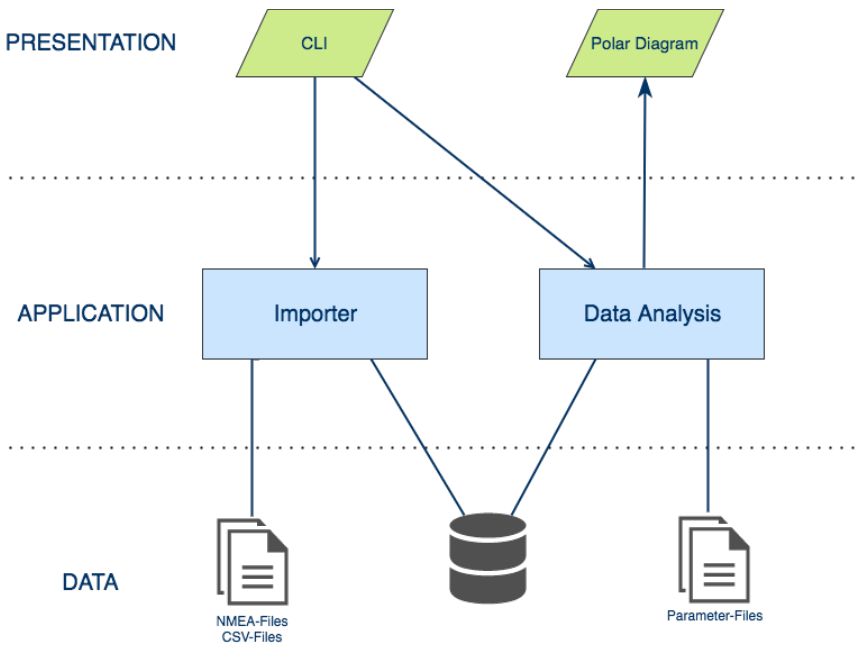
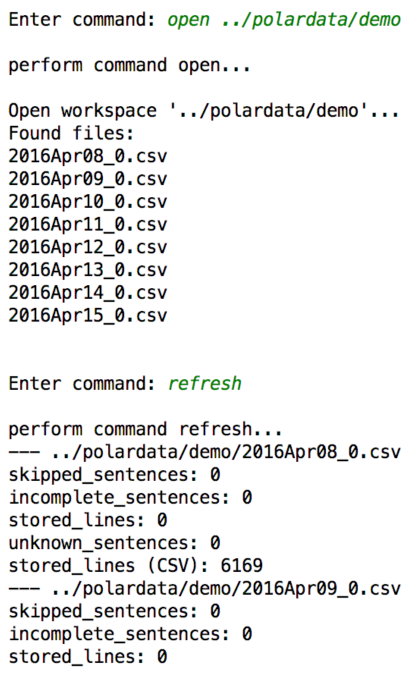
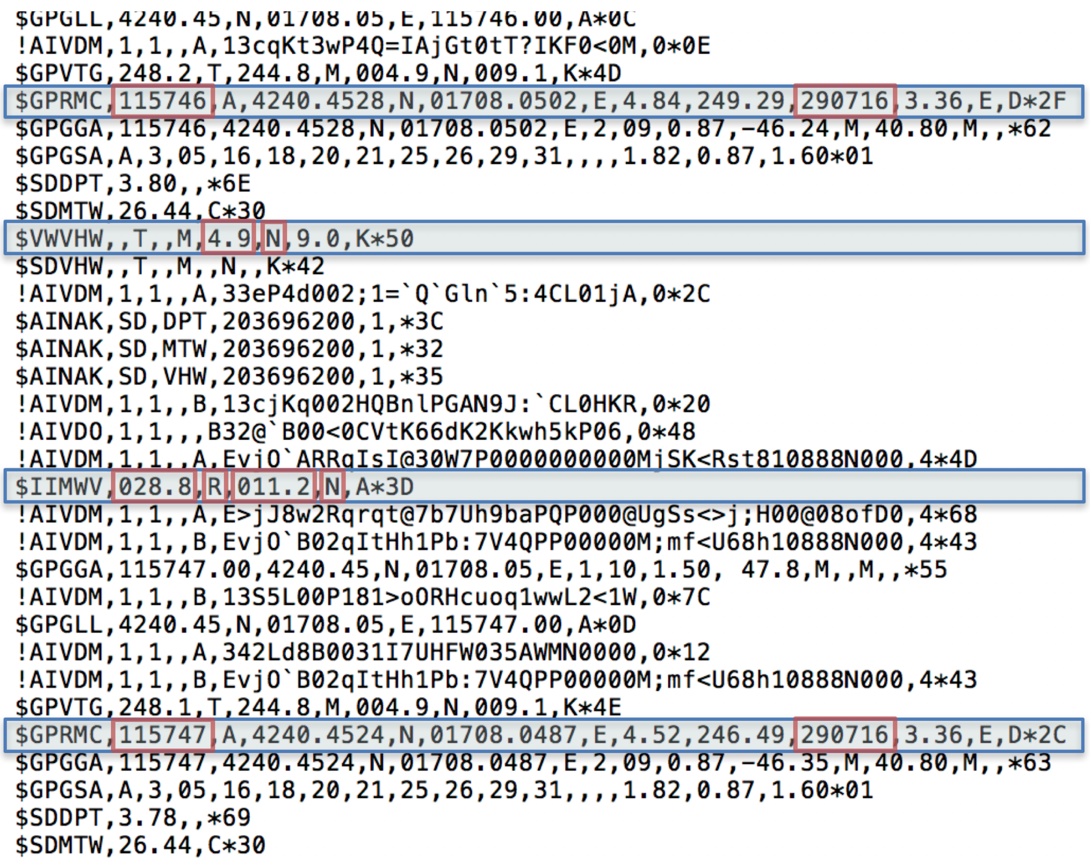


Figure 4.10.: 3-tier architecture applied in the developed software.



```
Enter command: open ../polardata/demo

perform command open...

Open workspace '../polardata/demo'...
Found files:
2016Apr08_0.csv
2016Apr09_0.csv
2016Apr10_0.csv
2016Apr11_0.csv
2016Apr12_0.csv
2016Apr13_0.csv
2016Apr14_0.csv
2016Apr15_0.csv


Enter command: refresh

perform command refresh...
--- ../polardata/demo/2016Apr08_0.csv
skipped_sentences: 0
incomplete_sentences: 0
stored_lines: 0
unknown_sentences: 0
stored_lines (CSV): 6169
--- ../polardata/demo/2016Apr09_0.csv
skipped_sentences: 0
incomplete_sentences: 0
stored_lines: 0
```

Figure 4.11.: Command line interface of the developed software.



```
$GPGLL,4240.45,N,01708.05,E,115746.00,A*0C
!AIVDM,1,1,,A,13cqKt3wP4Q=IAjGt0tT?IKF0<0M,0*0E
$GPVTG,248.2,T,244.8,M,004.9,N,009.1,K*4D
$GPRMC,115746,A,4240.4528,N,01708.0502,E,4.84,249.29,290716,3.36,E,D*2F
$GPGGA,115746,4240.4528,N,01708.0502,E,2,09,0.87,-46.24,M,40.80,M,,*62
$GPGSA,A,3,05,16,18,20,21,25,26,29,31,,,,1.82,0.87,1.60*01
$SDDPT,3.80,,*6E
$SDMTW,26.44,C*30
$VWVHW,,T,,M,4.9,N,9.0,K*50
$SDVHW,,T,,M,,N,,K*42
!AIVDM,1,1,,A,33eP4d002;1=`Q`Gln`5:4CL01jA,0*2C
$AINAK,SD,DPT,203696200,1,*3C
$AINAK,SD,MTW,203696200,1,*32
$AINAK,SD,VHW,203696200,1,*35
!AIVDM,1,1,,B,13cjKq002HQBnlPGAN9J:`CL0HKR,0*20
!AIVDO,1,1,,,B32@`B00<0CVtK66dK2Kkwh5kP06,0*48
!AIVDM,1,1,,A,EvjO`ARRqIsI@30W7P0000000000MjSK<Rst810888N000,4*4D
$IIMWV,028.8,R,011.2,N,A*3D
!AIVDM,1,1,,A,E>jJ8w2Rqrqt@7b7Uh9baPQP000@UgSs<>j;H00@08ofD0,4*68
!AIVDM,1,1,,B,EvjO`B02qItHh1Pb:7V4QPP00000M;mf<U68h10888N000,4*43
$GPGGA,115747.00,4240.45,N,01708.05,E,1,10,1.50, 47.8,M,,M,,*55
!AIVDM,1,1,,B,13S5L00P181>oORHcuoq1wwL2<1W,0*7C
$GPGLL,4240.45,N,01708.05,E,115747.00,A*0D
!AIVDM,1,1,,A,342Ld8B0031I7UHFW035AWMN0000,0*12
!AIVDM,1,1,,B,EvjO`B02qItHh1Pb:7V4QPP00000M;mf<U68h10888N000,4*43
$GPVTG,248.1,T,244.8,M,004.9,N,009.1,K*4E
$GPRMC,115747,A,4240.4524,N,01708.0487,E,4.52,246.49,290716,3.36,E,D*2C
$GPGGA,115747,4240.4524,N,01708.0487,E,2,09,0.87,-46.35,M,40.80,M,,*63
$GPGSA,A,3,05,16,18,20,21,25,26,29,31,,,,1.82,0.87,1.60*01
$SDDPT,3.78,,*69
$SDMTW,26.44,C*30
```

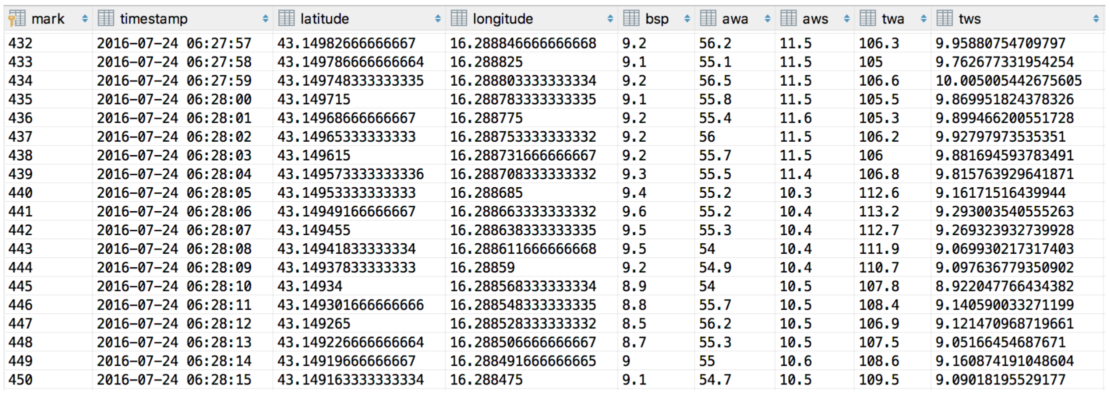
Figure 4.12.: Excerpt of a NMEA-File.

Read NMEA-Data

The data that was collected with MPXConfig2 is available in NMEA-Format spread to several files. A lot of information were collected that is not useful for the research problem. Each of the files was parsed using the library PyNMEA (section 4.4.4). It validates the checksum of the NMEA-sentences and splits it into the identifiers and data fields as defined in section 4.2.

To link variables together that happened at the same time, a reference identifier has to be selected. The identifier "GPRMC" was chosen which stands for the "Recommended Minimum Navigation Information" the GPS sends out periodically. This has the advantages that it is sent in a short time interval and the instant of time of the measurement is known. Then, all required variables between two GPRMC sentences are collected and marked with the same timestamp. In the excerpt in Figure 4.12 these used variables are marked in red.

The method of storing is discussed in the following subsection.



mark	timestamp	latitude	longitude	bsp	awa	aws	twa	tws
432	2016-07-24 06:27:57	43.14982666666667	16.288846666666668	9.2	56.2	11.5	106.3	9.95880754709797
433	2016-07-24 06:27:58	43.149786666666664	16.288825	9.1	55.1	11.5	105	9.762677331954254
434	2016-07-24 06:27:59	43.149748333333335	16.288803333333334	9.2	56.5	11.5	106.6	10.005005442675605
435	2016-07-24 06:28:00	43.149715	16.288783333333335	9.1	55.8	11.5	105.5	9.869951824378326
436	2016-07-24 06:28:01	43.14968666666667	16.288775	9.2	55.4	11.6	105.3	9.899466200551728
437	2016-07-24 06:28:02	43.14965333333333	16.288753333333332	9.2	56	11.5	106.2	9.92797973535351
438	2016-07-24 06:28:03	43.149615	16.288731666666667	9.2	55.7	11.5	106	9.881694593783491
439	2016-07-24 06:28:04	43.149573333333336	16.288708333333332	9.3	55.5	11.4	106.8	9.815763929641871
440	2016-07-24 06:28:05	43.14953333333333	16.288685	9.4	55.2	10.3	112.6	9.16171516439944
441	2016-07-24 06:28:06	43.14949166666667	16.288663333333332	9.6	55.2	10.4	113.2	9.293003540555263
442	2016-07-24 06:28:07	43.149455	16.288638333333335	9.5	55.3	10.4	112.7	9.269323932739928
443	2016-07-24 06:28:08	43.14941833333334	16.288611666666668	9.5	54	10.4	111.9	9.069930217317403
444	2016-07-24 06:28:09	43.14937833333333	16.28859	9.2	54.9	10.4	110.7	9.097636779350902
445	2016-07-24 06:28:10	43.14934	16.288568333333334	8.9	54	10.5	107.8	8.922047766434382
446	2016-07-24 06:28:11	43.149301666666666	16.288548333333335	8.8	55.7	10.5	108.4	9.140590033271199
447	2016-07-24 06:28:12	43.149265	16.288528333333332	8.5	56.2	10.5	106.9	9.121470968719661
448	2016-07-24 06:28:13	43.149226666666664	16.288506666666667	8.7	55.3	10.5	107.5	9.05166454687671
449	2016-07-24 06:28:14	43.14919666666667	16.288491666666665	9	55	10.6	108.6	9.160874191048604
450	2016-07-24 06:28:15	43.149163333333334	16.288475	9.1	54.7	10.5	109.5	9.09018195529177

Figure 4.13.: Database schema with some sample values.

Persistence

For easier and faster access to the data, the content of the NMEA files is imported into a database. A file-based SQLite database is chosen and its schema is illustrated in Figure 4.13. It will be automatically created in the workspace. All measured data with the same timestamp are stored in one database row. The columns for true wind speed (TWS) and true wind angle (TWA) are calculated later on during the normalization described on page 61.

If there are multiple values for a variable the mean is taken. If one of the required inputs of a database entry is missing, it can be omitted due to the exceeding amount of data.

According to Hwaci, 2017 SQLite is the most widely used database engine. It has support for the data types *null*, *integer*, *real*, *text* and *blob* and uses a single file for storage. The advantage is that no server is needed and its support on mobile devices with iOS and Android. It also deals as intermediate format. To support data of different kind, it just need to be converted to SQLite, which is even in the public domain. As an example a CSV-Importer was written to fill the database in addition to the NMEA-Parser.

Feature Overview

The developed software has an extensive feature set. All required modes of operation for the analysis were implemented and are available via the CLI. The available commands are described in Table 4.2.

Table 4.2.: Overview of the implemented commands.

`open [path]`	Set active workspace, create and load database. The path should lead to the recorded NMEA-files and is relative.
`close`	Close the active workspace
`refresh`	Load content from NMEA-files to the SQLite database.
`drop`	Clear the database of the active workspace.
`help`	Prints the documentation.
`quit`	Quit the program.
`load_configuration [path]`	Load a configuration file for the data analysis. Path is relative
`select_logbook`	Select data from manually filled logbook parameters. e.g. between two timestamps
`select_sail_finder [angle]`	Select data of a specific sail from an algorithm. The process is described in section 5.2.3.
`select_all`	Remove data limitations.
`trend_speed`	Generate trend diagram of boat speed (BSP) and TWS combined.
`trend_angle`	Generate trend diagram of TWA.
`trend`	Generate trend diagram of BSP, TWS and TWA combined.
`histogram`	Generate histograms of BSP, TWS and TWA.
`boxplot`	Generate box plot of BSP, TWS and TWA.
`occurrences`	Generate a heat map in a polar diagram of the frequency of data points at narrow areas of wind angle and wind speed.
`fit 2D_speed`	Perform model fitting of BSP and TWS. The used regression technique can be easily replaced.
`fit 2D_angle`	Perform model fitting of BSP and TWA.
`fit 3D`	Perform model fitting of BSP, TWS and TWA, with and without an established weight function.

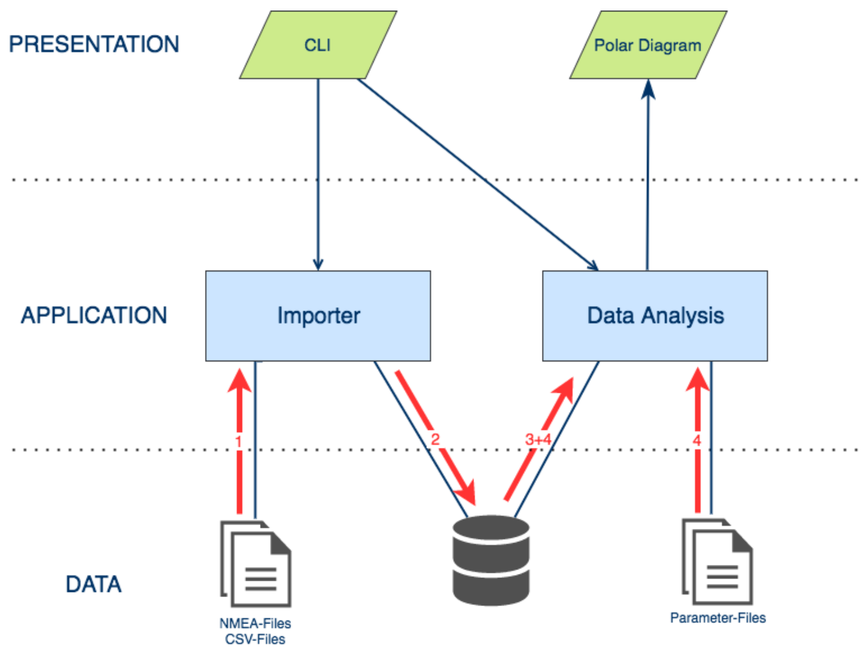


Figure 4.14.: Data flow of the polar diagram generator.

Sample Analysis

In this subsection an example of the usage of the program is given. It describes the steps that are necessary to generate a polar diagram from raw data.

1. `open ./folder/with/rawdata`
2. `refresh`
3. `select_sail_finder 60`
4. `fit 3D`
5. `quit`

The data flow of these commands 1 to 5 is illustrated with red arrows in Figure 4.14. The modules *Importer* and *Data Analysis* are trigger by the CLI with all necessary parameters.

`open ./folder/with/rawdata` This command is handled in the file `workspace.py`. It creates the database, a sample configuration file and required output folders.

`refresh` This command drops old content from the database and imports all raw data files from the workspace folder into the database. The classes `CSVImporter` and

NmeaImporter are responsible to extract the required information. After all data are inserted into the database, the script import_postprocessor.py is called to fill columns of the database that can be derived from the available information as described in subsection 2.1.2.

sail_finder 60 This command triggers the data selection algorithm described in section 5.2.3. It is implemented in sail_finder.py and analyzes the database to find ranges from that sail. The ranges that were found are used in all further commands.

fit 3D The modeling of the polar diagram is triggered with this command. In the file fit_3D.py the data is fetched from the database. It is limited to the ranges defined in the configuration file and a previously executed sail_finder command. Thus in this sample the resulting polar diagram is generated for one sail only. It can be found in the output folder of that run, similar to ./folder/with/rawdata/1970-01-01T0000/plots/.

This command can be configured with the parameter file. Beside the parameters for limiting the range of the data, one useful parameter is show_plots. It allows to view the generated plots to change zoom level and view angle. Another parameter is perform_animation_plots which, if set to *true*, triggers the generation of a short animation which visualizes the 3-dimensional fit.

quit When this command is entered, the program performs a clean exit.

4.4.3. Software Techniques

This subsection has the focus on main software design principles. These were applied during the development of the software to perform the data analysis.

Testing

Testing can be made in varying types. Component tests verify a small part of the software independent from the remaining parts whereas integration tests examine the connected parts of a software. In Python the standard library *unittest* has a similar interface to the well known testing libraries of other programming languages JUnit, nUnit or CppUnit.

The term *mocking* in conjunction with testing is a technique to change the behavior of a class during a test-run.

Strategy Design Pattern

Design patterns in software technology are best practices for common problems. A definitive book is *Design Patterns: Elements of Reusable Object-oriented Software* by the so-called *Gang of Four*.

The behavioral design pattern *strategy* is worth to mention because it is used at multiple locations in code and contributes to the easy extensibility of the program. An interface of an abstract function is designed. The concrete implementation of that function is irrelevant.

In case of this project the implementation of the different models and the type of regression was made interchangeable with this design pattern.

4.4.4. Third Party Dependencies

This subsection gives an overview of the main libraries that were used during this project.

Cython

According to Behnel et al. with Cython it is possible to embed faster implementations of algorithms written in C-Code into Python with marginal performance loss. The extension is open source and freely available under Apache 2.0 (*Cython License* 2009).

SciPy

SciPy is a set of scientific computing tools for Python. It contains amongst others *NumPy* for numerical computations, the *SciPy Library* for optimization or statistics and *Matplotlib* for plotting (Jones, Oliphant, Peterson, et al., 2001–). Most of the bundled libraries are licensed under Berkeley Software Distribution (BSD).

Scikit-Learn

A library for machine learning that is connected to SciPy section 4.4.4 is Scikit-learn (Pedregosa et al., 2011). It was started as a Google Summer of Code project in the year 2007 and currently available in version 0.19.0. The license of this library is BSD.

Its distinctiveness is the clean API that allows varying regressors to be interchangeable and serial executable in an easy way. The interface has the following signature (Buitinck et al., 2013):

```
def fit(self, X, y)
def predict(self, X)
```

Custom implementations of regression techniques just need to inherit from *BaseEstimator* and implement these two methods. This was made to add support for orthogonal distance regression (ODR) with non-linear models to Scikit-learn.

Pandas

Pandas is a Python library for handling labeled data structures. It was developed to perform data analysis and its features support during the preparation, modeling and visualization of data. It is open source and licensed under BSD.

PyNMEA

It was mentioned in section 4.2 that the data is available as consecutive NMEA sentences. To verify the sentences and extract the necessary information the library PyNMEA was created. It was originally developed by Becky Lewis but has several forks in the meantime. The library is open source and distributed under Massachusetts Institute of Technology (MIT) license.

GPSBabel

GPSBabel is a software to convert and transform GPS-tracks from several GPS receivers. They often use a format that is slightly different according to ordering and delimiters. Robert Lipe started this tool in the year 2002 to make the output of these receivers interchangeable. It is available under the GNU Public License.

5. Use Case

This chapter describes the application of the previously presented methods to address the research question of the thesis. It is structured similar to the steps of *data analysis* in chapter 3.

5.1. Problem Definition

Starting with the first step of data analysis is the definition of the problem that should be solved: A polar diagram of a sailing boat should be generated. As described in chapter 2 the wind is driving the sailing boat. That means that the boat speed is influenced by wind speed and wind angle which can be seen as a function in Equation 5.1.

$$BSP = f(TWS, TWA) \tag{5.1}$$

To permeate deeper into the matter an outline of the current situation on sailing boats is described in the following subsection.

5.1.1. Overview

Nowadays on most sailing boats there are several measurement devices. The normal data flow is diagrammed in Figure 5.1 where most boat instruments are used for display the current measured value only. The most common devices are wind gauge (subsection 4.1.1), speed log (subsection 4.1.2) and GPS (subsection 4.1.3).

The goal is to generate a polar diagram (subsection 2.3.1) in a less intricate way. Basis is the general assumption that performance that was reached under certain conditions can be reached again under the same conditions at an later date. This leads to the approach to record the performance over a time range, process the data, fit a model to the data and generate polar diagrams.

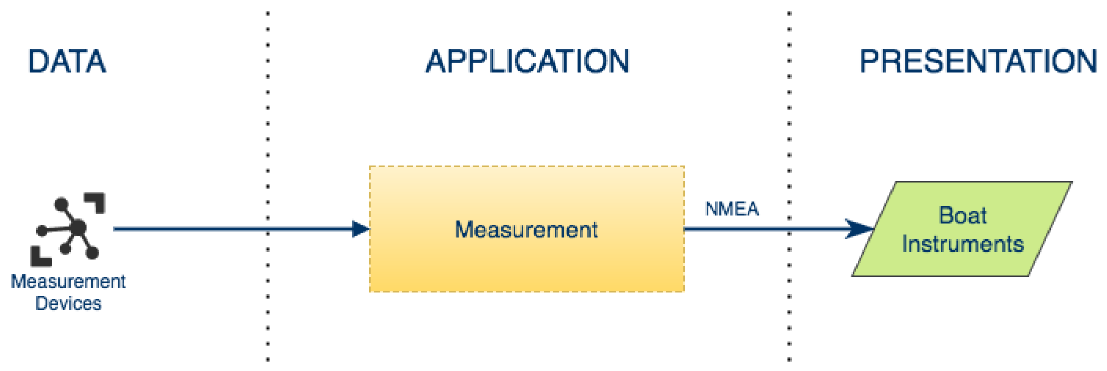


Figure 5.1.: Data flow on current sailing boats in a 3-tier architecture (selfmade).

During the test run sensors, as described in section 4.1, are logging the value of wind speed, wind angle and boat speed. For this analysis other factors like the current sea are not explicit measured. A reason for this decision is that suitable measurement devices are rarely installed on sailing boats. Furthermore the concrete values of forces that are generated by the wind are not determined and it is assumed that the crew always tries to reach the best performance under the given conditions.

With the scheme sketched in Figure 5.2 the extension module that generates polar diagrams out of a set of measured data is easy connect-able to the existing system on common boats.

For the visualization of the performance polar diagram, the accuracy of the stated value is needful. Areas with a high amount of data points with low variance lead to very high confidence. On the contrary, on ranges with only a few data points no statement should be made at first but it should be investigated if the boat performance of such a range can be interpolated from areas with high confidence.

5.1.2. Technology Screening

When handling with measured data two main techniques can be made. One is to visualize the data without making any assumptions on the data. A standard way to perform this is interpolation with lines, polynomials or splines. Methods with good performance exist and lead to fast results.

The other method is to make assumptions on the data from physical background and observation. This is called *data analysis* (chapter 3) and consists of a multitude of different kinds. Some of them are: linear regression, spatial regression, stochastic processes and time series.

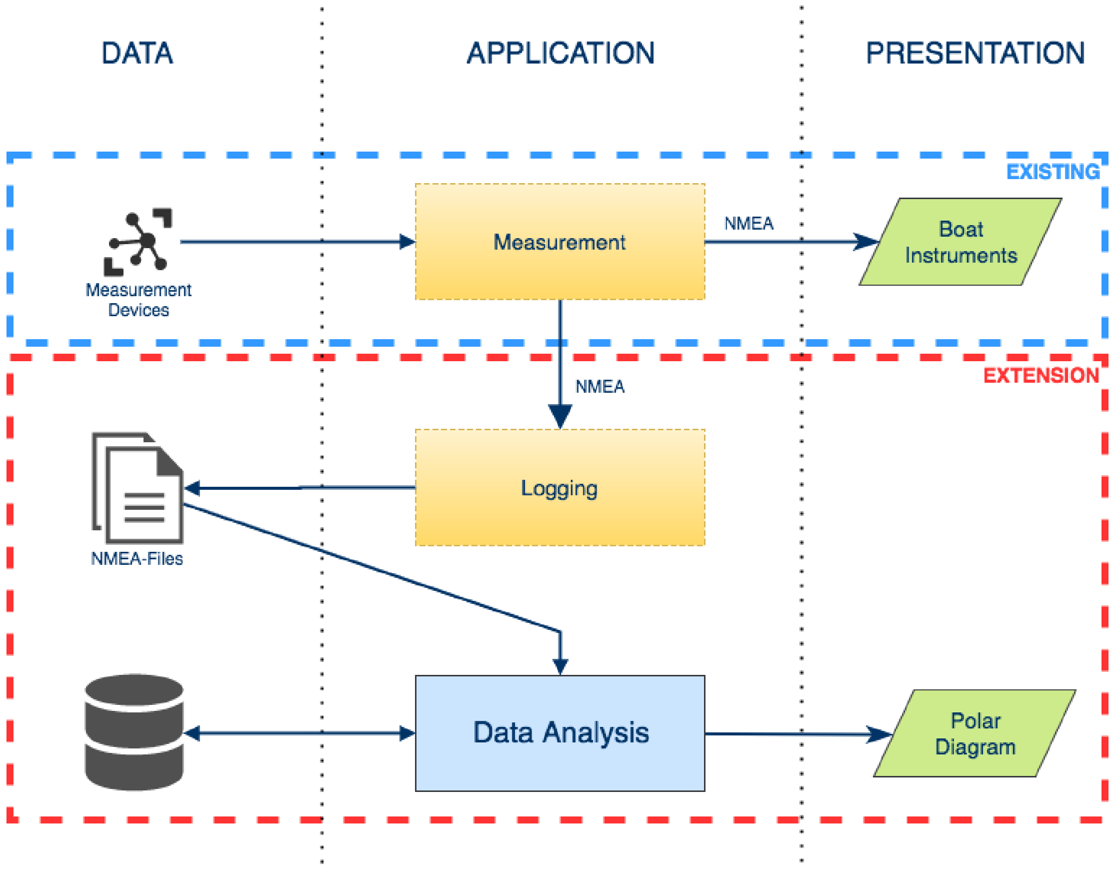


Figure 5.2.: Scheme of the extension to the existing system in a 3-tier architecture (selfmade).

$$\begin{tikzcd} BSP = f(TWS) \arrow[dr] & & BSP = f(TWA) \arrow[dl] \\ & BSP = f(TWS, TWA) & \end{tikzcd}$$

Figure 5.3.: The procedure to develop a 3-dimensional model.

For this problem regression analysis (subsection 3.4.1) was chosen. The advantage over interpolation is that the result consists of a model which can be used to extrapolate data. That means that areas with a low number of data points can be derived from other areas.

A disadvantage is the complex development of models that have the right amount of flexibility to represent the data. The detailed account of the approach of generating models for polar diagrams is given in the following section.

5.1.3. Proceeding

This section has stated that a 3-dimensional model should represent the influence of the wind to the boat speed. It is now necessary to explain the course of developing such a complex model. The process is split into three steps. Each of the steps embrace a data analysis of a subset of the data and feature space, which simplifies the analysis. At the beginning, a correlation of boat speed (BSP) and true wind speed (TWS) is inquired. In a second step, the relationship of the true wind angle (TWA) to the BSP is analyzed. If successful, both steps will return 2-dimensional models as output.

The most promising models are used to develop models for polar diagrams that take the 3 features BSP, TWS and TWA into account. This procedure is illustrated in Figure 5.3.

5.2. Data Collection and Organization

After the problem is well defined, the following step is to identify how the data is collected that should be analyzed. For the problem of generating polar diagrams, suitable data of performance measurements on sailing boats are not publicly available, thus some experiments to generate such data are required. This section examines the necessary preparation of the data to have an organized access to them during the data

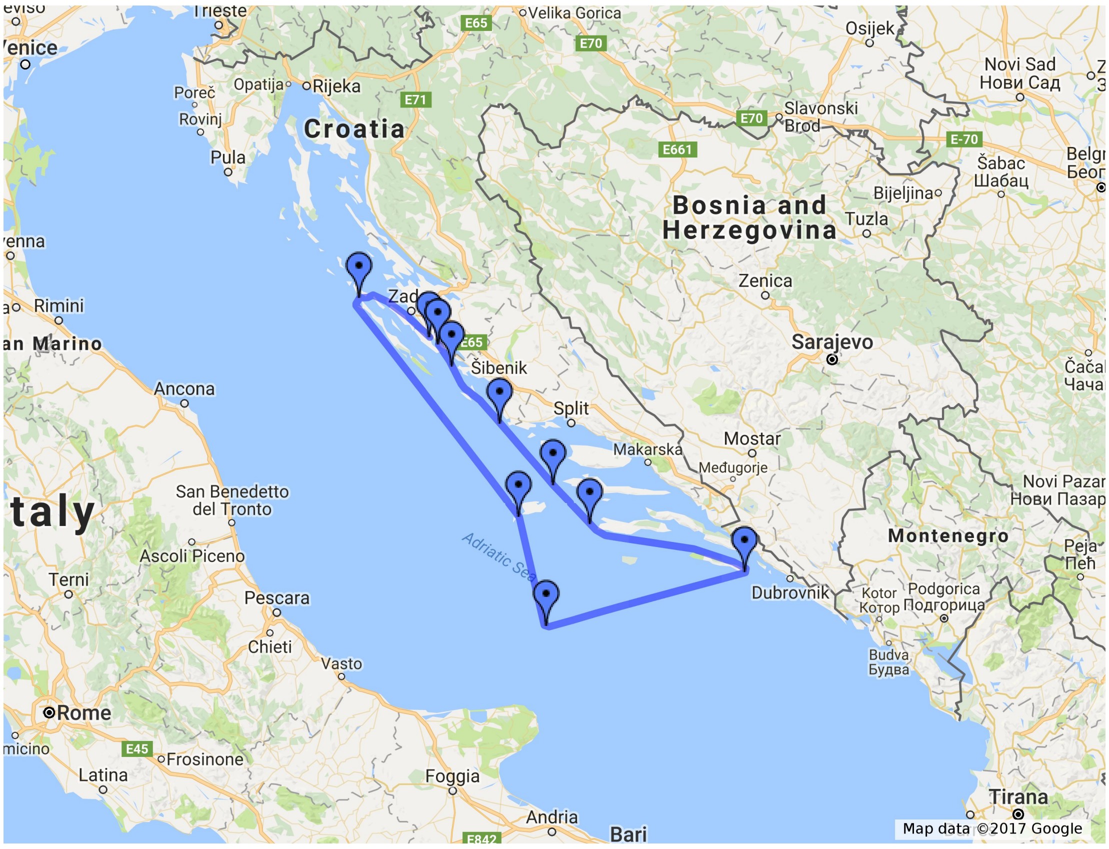


Figure 5.4.: The circuit of the Round Palagruža Cannonball (*Die Route des Round Palagruža Cannonballs* 2010).

analysis. Details about the realization of the proceedings in software are described in section 4.4.

5.2.1. Measurement Environment

To meet the demand that the crew sails the ship with the highest possible performance the data were measured during a regatta. The Round Palagruža Cannonball (RPC) in April 2016 was chosen. The regatta takes place in Croatia and has its start in Biograd. The course is shown on a map in Figure 5.4 and has the required waypoints Dugi Otok, Gate Vis/Bisevo, Palagruža, Mljet, Hvar, Mulo and finishes in Biograd again. This are about 360 nautical miles on map which are sailed non stop.

The data were taken on a Bavaria 40S and the ship was called “Jasmin”. The boat has a hull length of 12.426 meters (*ORC Club Certificate Bavaria CR 40S* 2012) and a waterline length of 10.75 meters (*Bavaria Cruiser 40S* 2016). Given the equation of subsection 2.3.4 the maximum hull speed is about 8 knots.

The wind gauge to measure wind speed and direction and the common log to measure boat speed were manufactured by B&G. The measurements were transmitted with the older NMEA-0183 bus to a Miniplex2 and over a virtual COM port to the software MPX-Config (*MPX-Config2 for MiniPlex-2* 2015). The exported NMEA-Files were examined with a custom-made software for analyzing data from sailing boats, which is described in section 4.4.

5.2.2. Normalization

After Importing the data and persisting it in the database, the data is automatically normalized. This ensures a homogeneous basis for all future calculations. The measured wind angle is delivered in two different ranges, dependent on the measurement device. One reaches from -180 to $+180$ degrees relative to the bow and the other from 0 to 360 degrees clockwise starting at the bow. When dealing with real-life data, such issues should be resolved. In this case the data was normalized by using the range from -180 to $+180$.

The wind gauge is installed at the top of the mast of the boat so it measures apparent wind speed (AWS) and apparent wind angle (AWA) dependent on the BSP. When using regression it is important that the independent variables are uncorrelated. With the method described in subsection 2.1.2, the true wind conditions were derived from the measured apparent wind and boat speed. Afterwards, all calculations were made using true wind conditions.

5.2.3. Data Selection

Data were measured continuously including areas that are not interesting during an analysis. In this subsection the mechanism how data was left out is discussed.

Expected High Performance

The analysis after the sailing turn shows that data consists of over $400,000$ data rows. Only the part during the regatta is taken for the following analysis to exclude points that may not be representable for full performance of the crew. That results in a data

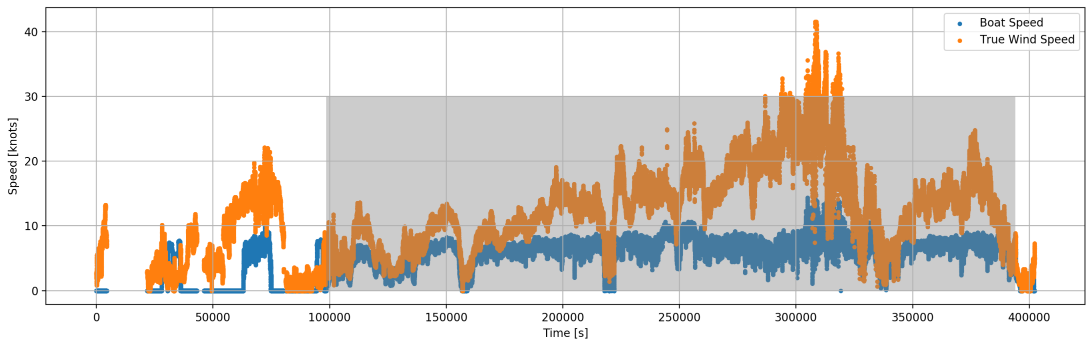


Figure 5.5.: Trend of BSP and TWS during the testing period.

set of 295,597 entries beginning with May 11, 2016 12:00 nonstop to May 14, 2016 23:51 when the boat crossed the finish line. This timespan is marked in Figure 5.5.

In this figure it is illustrated that during the regatta there was barely no wind sometimes. Offshore Racing Congress (ORC) performance diagrams start at 6 knots. For further calculations values with a boat speed or wind speed smaller than 2 knots were omitted because there are a lot of turbulences and results in that speed-range are not interesting at all.

Sail Selection

It was mentioned in section 2.3 that there is a separate polar diagram for every sail. Thus the data has to be assigned to the belonging sail and the analysis has to made for each sail separately. During the regatta a standard logbook was written which is not accurate enough. Therefor an algorithm was developed to identify all values of a particular sail. It uses the fact that sails are operated at a narrow range of wind angle.

It works as follows: at the beginning a common true wind angle of a sail is estimated. For a jib this could be 60 degrees. The algorithm chronological calculates the mean of the sailed wind angle from an interval of 20 seconds. If the mean is within a range around the defined angle of the sail, it is considered that the data point was recorded with that sail. If a sail is active, the mean of the following interval is compared not with the defined angle but with its preceding interval. Thus smooth adjustments of the course can be considered as the same sail even if the sail is operated far outside its estimated optimal angle. A range from about 40 to 120 degrees can be correctly identified as a jib which is illustrated in Figure 5.6.

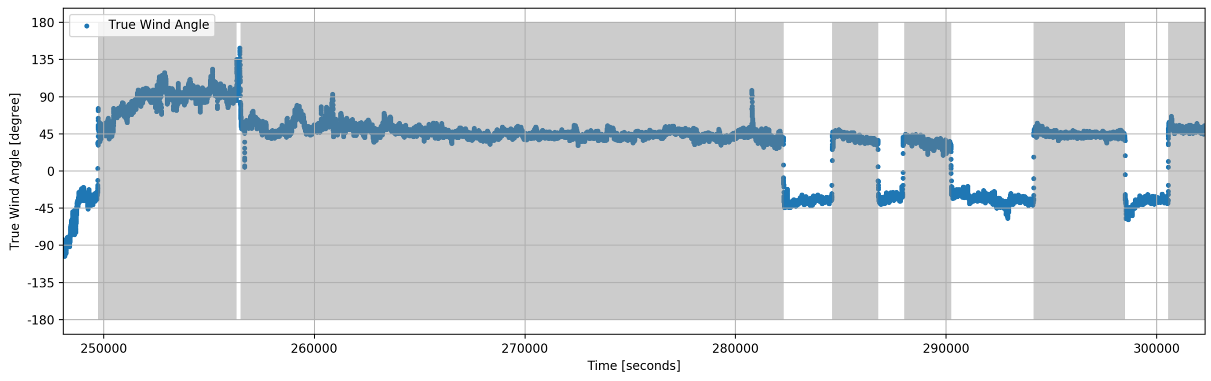


Figure 5.6.: Trend of TWS with identified ranges while sailing with a jib.

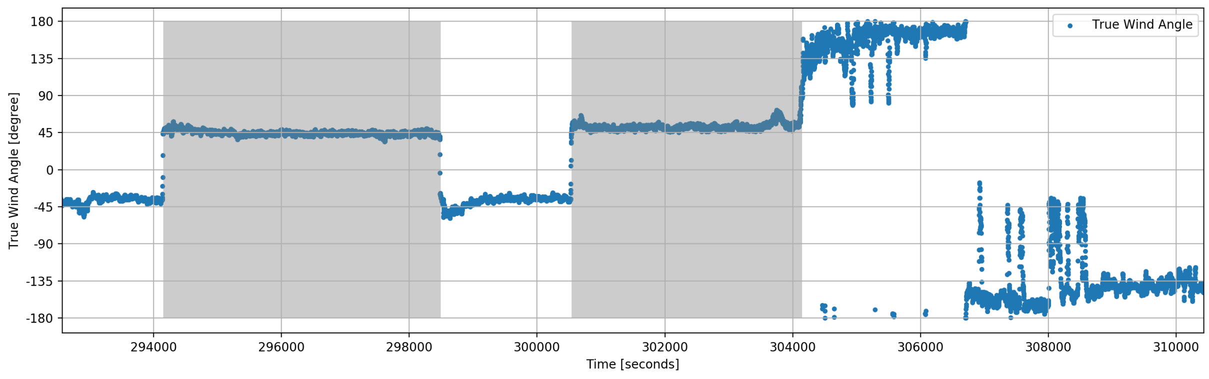


Figure 5.7.: Trend of TWS with identified ranges during the change from a jib to a spinnaker.

The algorithm only selects one bow to take care of a common misconfiguration (cf. subsection 4.1.1) of the wind gauge, later on. If this is not desired the areas for $+60$ degrees and -60 degrees can simply be added. In Figure 5.7 the algorithm correctly identifies the change to a different sail.

5.3. Correlation of Boat Speed and Wind Speed

As mentioned previously, the concept of dividing a complex problem into smaller problems is used for the analysis. In the first step the relationships of only two variables is analyzed and the result is used to gather a model that solves the problem of generating a polar diagram. At the start of the analysis the correlation of both values that represent speed is sought-after. Given the physical background in chapter 2, this will show the concrete influence of wind speed to the boat speed. This is defined with the following Equation 5.2.

$$BSP = f(TWS) \tag{5.2}$$

It was already discussed in chapter 2 that the boat speeds depends on wind speed and wind angle. Thus the influence of the wind angle should be diminished by choosing data points from an appropriate fixed wind angle.

5.3.1. Data exploration

In this subsection a general conspectus of the data to compare the both speed variables is shown. In Figure 5.5 the overall trend of boat speed and wind speed is illustrated. When taking a closer look at the trend in Figure 5.8 a direct proportional connection is revealed.

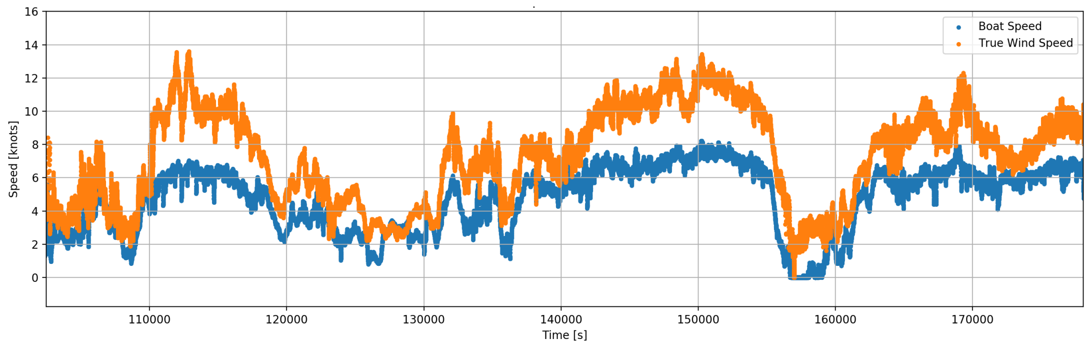


Figure 5.8.: A detailed view of the trend of BSP and TWS.

In Figure 5.9 a higher frequency of occurrences of close-hauled angles is evident. This matches with the field report of *Round Palagruža Cannonball, Medien-Info 3* 2016.

When applying the sail selection algorithm from section 5.2.3 to identify the operating angles of a specific sail the resulting histogram is illustrated in Figure 5.10.

There are two normal distributions around the center of -80 and -35 degrees. The area around those angles seem to be interesting and hence are examined in the following subsection. In Figure 5.11 and Figure 5.12 the measured data points at these angles are illustrated.

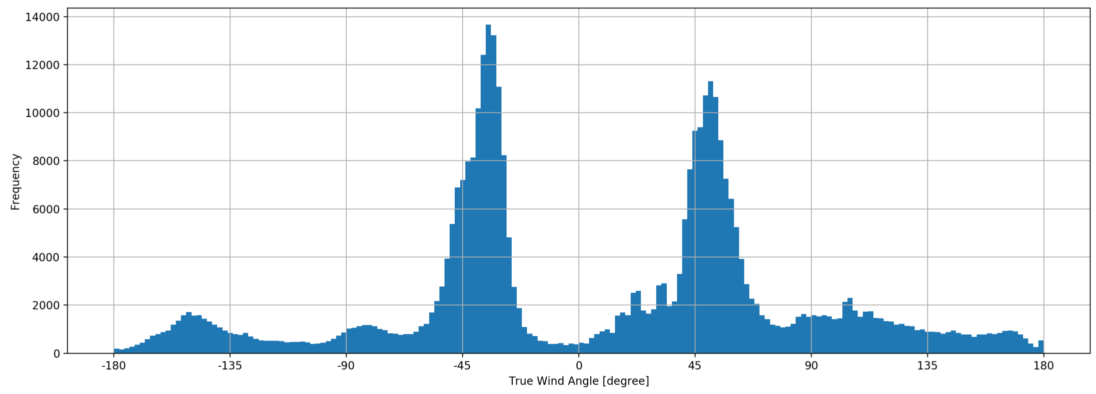


Figure 5.9.: Histogram of measured data points broken down by TWA.

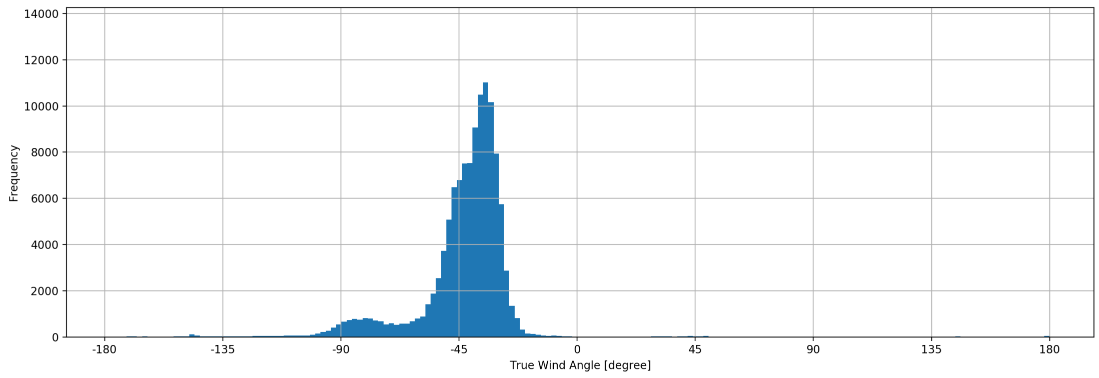


Figure 5.10.: Histogram of measured data points broken down by TWA. The data were limited to a Jib on the port-side bow.

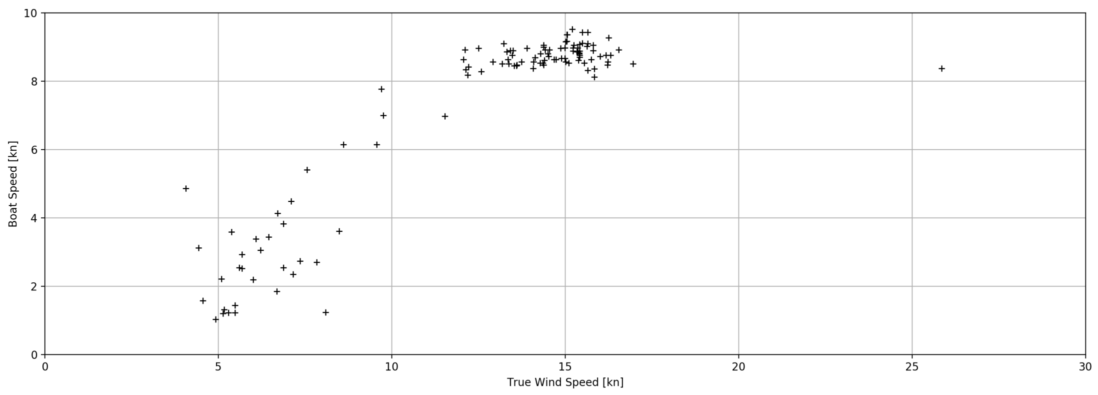


Figure 5.11.: Scatter plot of BSP and TWS at an TWA of -80 ± 0.5 degrees

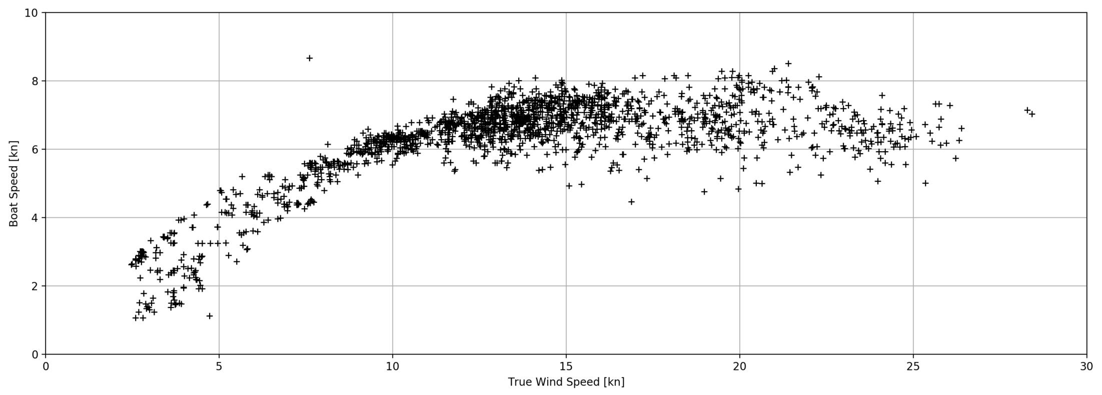


Figure 5.12.: Scatter plot of BSP and TWS at an TWA of -35 ± 0.5 degrees

5.3.2. Identification of methods, hypothesis and tests

Turning now to the *inferential statistic* to find a connection in the explored data. There is the intention to fit a line into the data points that represents the correlation of BSP and TWS.

The recorded data of the previously mentioned angles have a similar shape and seems to follow the same model. Thus this subsection focuses on the data of Figure 5.12 with an angle of -35 ± 0.5 degrees, because it contains a multiple amount of data points.

A common method is regression analysis (cf. subsection 3.4.1). In this case both variables have uncertainties that can't be neglected. Consequential the residuals have to be calculated using orthogonal distance regression (ODR) as mentioned in section 3.4.1.

In the Python code, to be expandable to future estimators, each estimator has to implement a pre-defined interface. The interface defined by Scikit-learn (section 4.4.4) was used. For nonlinear regression and orthogonal distance regression estimators of SciPy were used. They have a different API, thus a wrapper was created.

To find the best hypothesis, multiple models are trained simultaneously and compared. The tested models are defined in Table 5.1. Amongst others, Backhaus et al., 2016, p. 577 defines a nonlinear model called *Concave with Downturn* which is equal to a 2-degree polynomial with negative leading sign at the quadratic term.

All trained models have to be tested for validity in regard of under-fitting, over-fitting, homoscedasticity and non-autocorrelation. This is done with a scatter plot, a residual plot and the Durbin/Watson statistic as described in section 3.4.2.

Table 5.1.: Hypothesizes for the correlation of BSP and TWS. Most of the models are defined by Backhaus et al., 2016, p. 577.

2-degree polynomial 3-degree polynomial 4-degree polynomial 6-degree polynomial	$f(x) = \sum_{k=0}^{n} v_k * x^k$
Inverted Parabola	$f(x) = b + c * (x - a)^2$
Concave with Downturn	$f(x) = a + b * x - c * x^2$
Concave with Saturation Limit	$f(x) = c - a * e^{-b*x}$
Concave with Saturation Limit and Downturn	$f(x) = c - a * e^{-b*x} - d * x^2$
S-shaped with Saturation Limit	$f(x) = \frac{c}{(1 + e^{a-b*x})}$
S-shaped with Saturation Limit and Downturn	$f(x) = \frac{c}{1 + e^{a-b*x}} - d * x^2$
Gompertz-Model	$f(x) = c * e^{-a*b^x}$

Finally the best hypothesis of the measured data is chosen as proposed in subsection 3.4.3: The measured data were split into training and test set. The model with the smallest χ^2 on the test set can be considered as the best model.

5.3.3. Analysis and Interpretation

The plotted result of the tested models are illustrated in Figure 5.13.

The computed quality parameter of the individual model can be found in Table 5.2.

A more detailed plot of the S-shaped model with Saturation Limit and Downturn including a residual plot can be found in Figure 5.14. Its model with fitted parameters is defined in Equation 5.3

$$f(x) = \frac{1.1298}{1 + e^{1.5999 - 6.3780x}} - 0.5701x^2 \tag{5.3}$$

Interpretation Fitting about 6.000 data points with nonlinear orthogonal distance regression in 2 dimensions takes under 200 *ms*. Thus, using this method for a real-time feedback would be possible.

When it comes to visual model selection, the 6-degree polynomial obviously shows the phenomenon of *over-fitting* and therefor is too complex. The visual criterion for

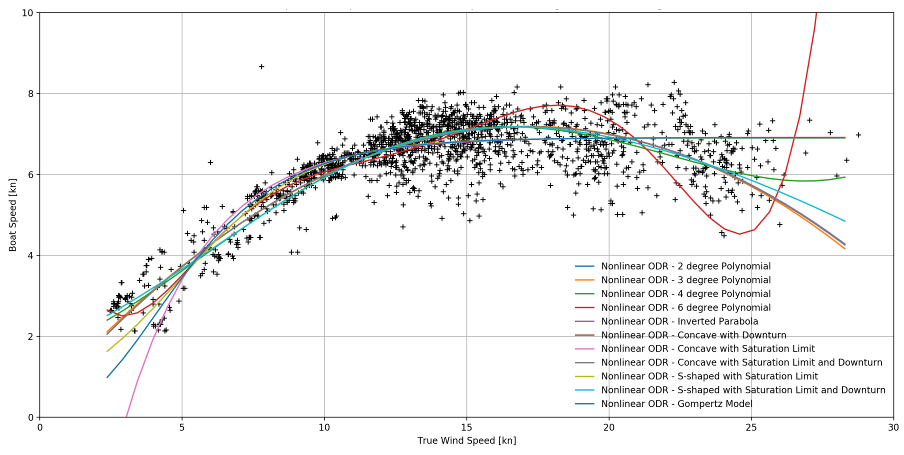


Figure 5.13.: Comparison of TWS vs. BSP at -35 ± 0.5 degrees with multiple models

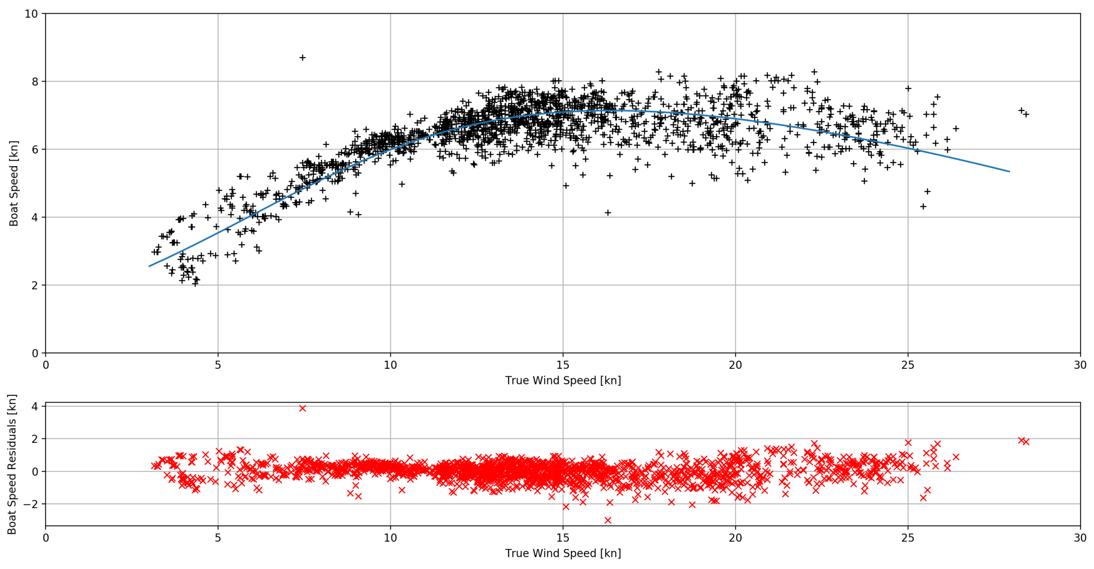


Figure 5.14.: Comparison of TWS vs. BSP at -35 ± 0.5 degree with a fitted S-shaped model with Saturation Limit and Downturn and its belonging residual plot.

Table 5.2.: Quality values of Orthogonal Distance Regression with varying models

Model	Runtime	Durbin/Watson	χ^2_{train}	χ^2_{test}
2 degree Polynomial	0.05 s	1.9827	108.05	68.39
3 degree Polynomial	0.09 s	1.9815	108.06	69.15
4 degree Polynomial	0.62 s	2.0103	109.06	67.01
6 degree Polynomial	8.34 s	2.0052	104.67	168.57
Inverted Parabola	0.11 s	1.9827	108.05	68.39
Concave with Downturn	0.03 s	1.9827	108.05	68.51
Concave with Saturation Limit	0.11 s	1.9824	140.01	129.28
Concave with Saturation Limit and Downturn	3.27 s	1.9828	108.08	68.54
S-shaped with Saturation Limit	0.11 s	2.0184	135.50	70.38
S-shaped with Saturation Limit and Downturn	0.15 s	1.9984	108.93	66.65
Gompertz Model	0.28 s	2.0116	137.52	78.81

exclusion of the remaining models are depleted, thus the χ^2 should be compared to choose the best model. The model with the lowest χ^2 on the test data is the S-shaped model with Saturation Limit and Downturn which will be examined in detail.

When taking a look at the residuals plot in Figure 5.14 it does not have the shape of a triangle hence the precondition of homoscedasticity holds. The result of the Durbin/Watson-Test (section 3.4.2) is 1.9984 which is close to 2 and means there is no autocorrelation. Therefore the fitted model can be treated as valid.

In Figure 5.14 the fitted line shows a decrease of the boat speed if the wind is stronger than 18 knots. That means there is not only a maximum speed but also a decrease of boat speed with increasing wind speed. The reasons could be a combination of larger waves and a modified lateral plan due to a different heeling.

Note: With reference to section 3.4.1, the result of regression methods can be quite different. Ordinary least squared (solution in one step) and nonlinear regression (iterative solution finding) lead to the same result but when using the correct orthogonal distance regression the result can deviate about 0.3 knots (Figure 5.15).

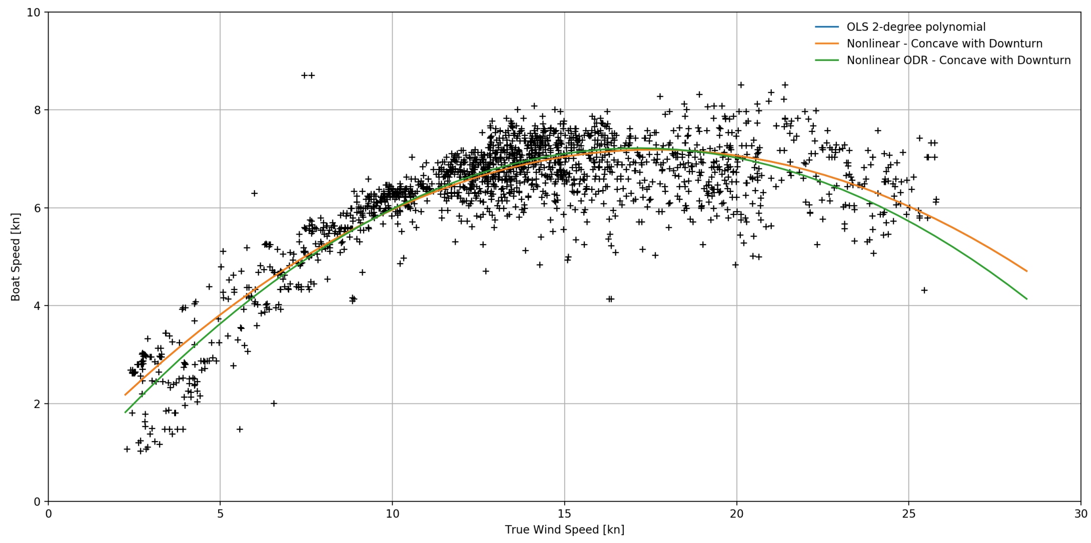


Figure 5.15.: Comparison of different regression methods.

5.3.4. Result

The resulting models of this regression seems to represent the data well. No further separated exploration of BSP and TWS should be necessary. However the χ^2 of several models is at close quarters so that no model can be identified as exclusive best model. Thus for the creation of a 3-dimensional model the following models were taken into account beside the model with the lowest χ^2:

- Concave with Downturn
- S-shaped with Saturation Limit
- Concave with Saturation Limit and Downturn
- S-shaped with Saturation Limit and Downturn

5.4. Correlation of Boat Speed and Wind Angle

In this section the influence of the wind angle to the boat speed is focused. The mathematical representation is stated in Equation 5.4.

$$BSP = f(TWA) \tag{5.4}$$

Similar to the previous section the focus is on the Jib and the influence of the wind speed has to be minimized. Therefor data from a narrow wind range are used for the

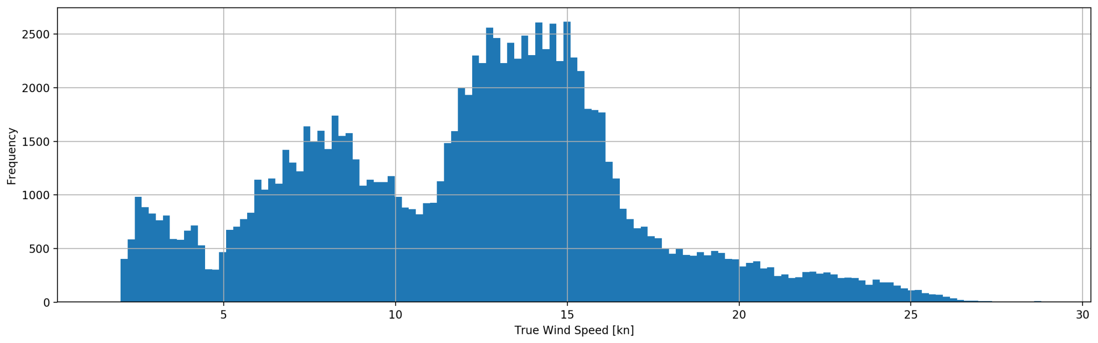


Figure 5.16.: Histogram of measured data points broken down by TWS.

analysis at this stage. The process of data analysis starts again with exploratory data analysis.

5.4.1. Data exploration

In Figure 5.16 a histogram of the data points that occurred at the various wind speeds is illustrated. It is observable that the rate of occurrences of the wind speed from 7 to 9 and 12 to 16 knots are higher. Around 8 knots are 11, 129 and around 14 knots TWS are 31, 848 data points. That are good amounts of data points to analyze.

In Figure 5.17 and Figure 5.18 the data points in that ranges are illustrated in a scatter plot.

5.4.2. Identification of methods, hypothesis and tests

For the combination of BSP and TWA the same preconditions like in the previous comparison of BSP and TWS hold. Thus a bivariate regression with orthogonal distance calculation should be applied on the data in slices of all available data. There is a greater difference in the shape illustrated in Figure 5.17 and Figure 5.18. So both slices are tested against the same hypotheses which is illustrated in Table 5.1.

5.4.3. Analysis and Interpretation

The plot of the fitted models around 8 and 14 knots TWS is illustrated in Figure 5.19 and Figure 5.20.

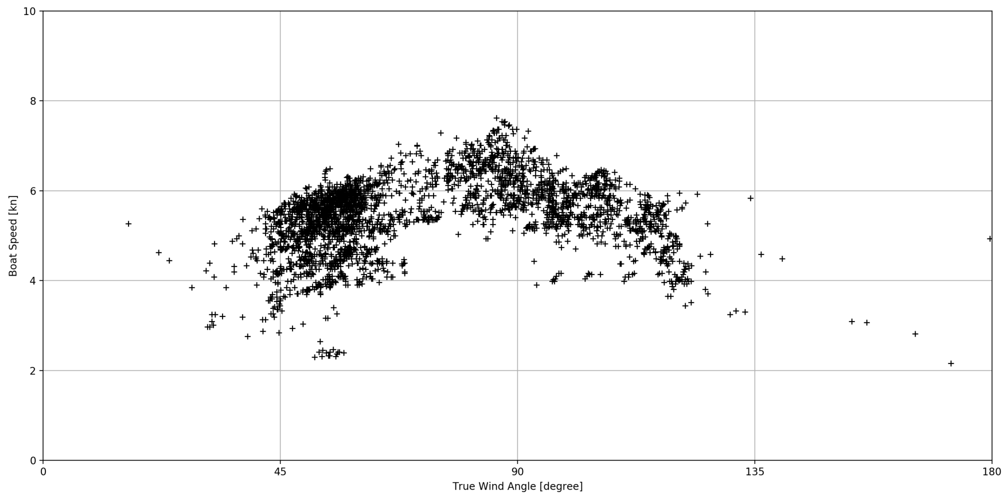


Figure 5.17.: Scatter plot of measured data points at a range from 7 to 9 knots TWS.

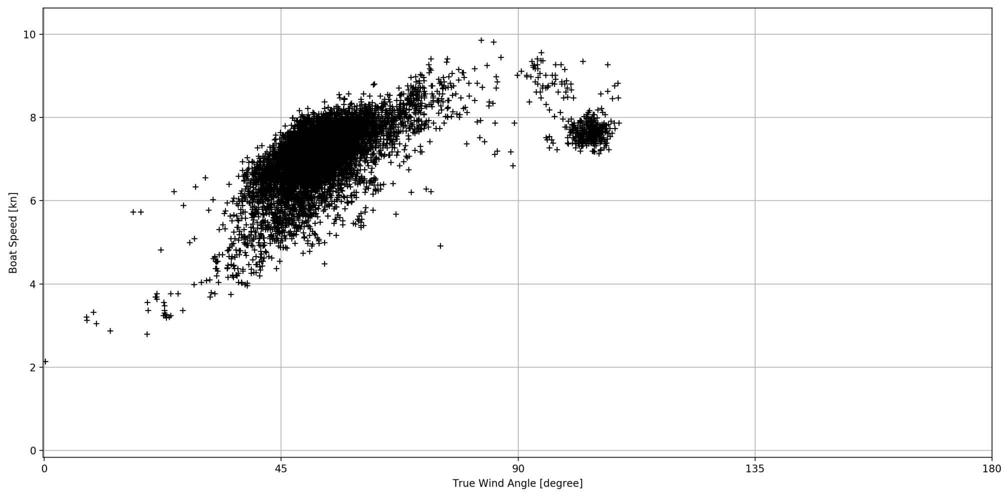


Figure 5.18.: Scatter plot of measured data points at a range from 12 to 16 knots TWS.

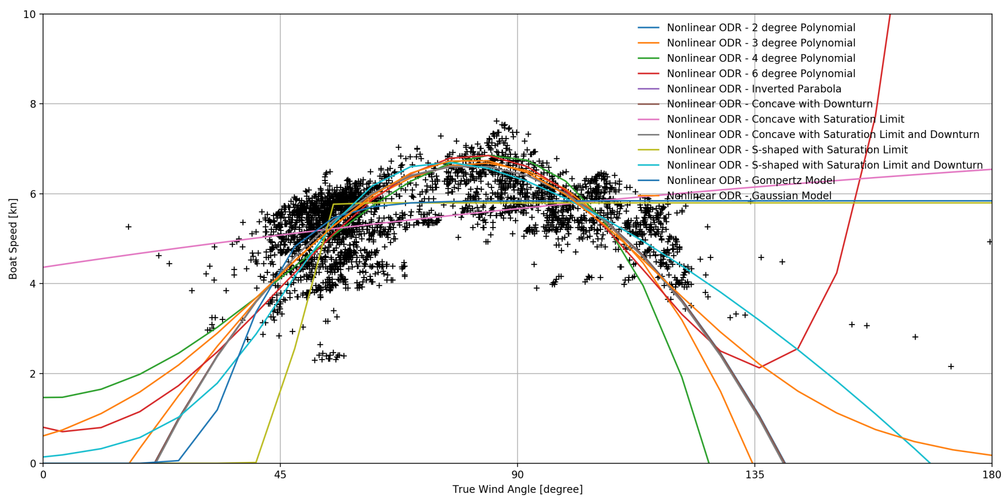


Figure 5.19.: Comparison of TWA vs. BSP from 7 to 9 knots TWS with multiple models

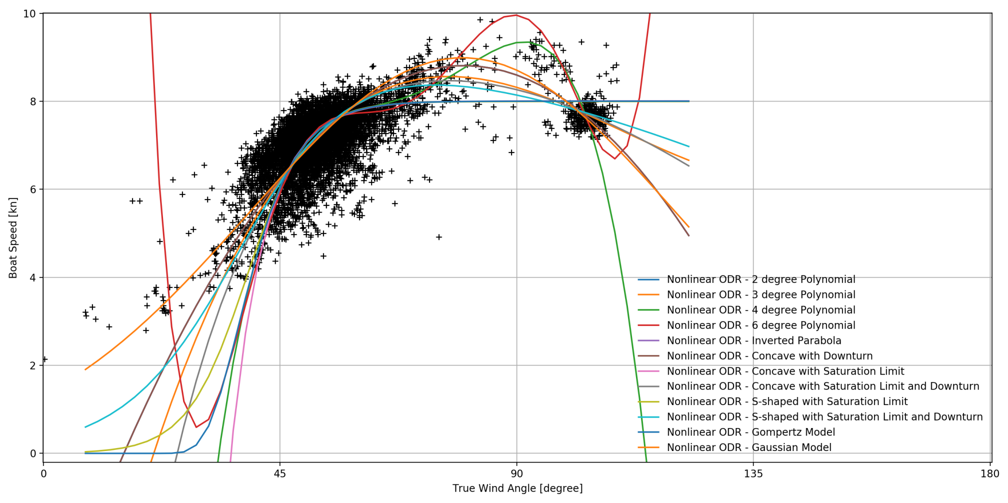


Figure 5.20.: Comparison of TWA vs. BSP around 14 knots TWS with multiple models

Table 5.3.: Quality values of ODR with varying models from 7 to 9 knots TWS

Model	Runtime	Durbin/Watson	χ^2_{train}	χ^2_{test}
2 degree Polynomial	0.50 s	1.9590	332.40	11345.56
3 degree Polynomial	0.50 s	2.0099	206.90	937.76
4 degree Polynomial	0.70 s	1.9790	165.97	1274.64
6 degree Polynomial	1.57 s	1.9423	131.04	492.77
Inverted Parabola	0.63 s	1.9592	332.10	11454.64
Concave with Downturn	0.37 s	1.9591	332.30	11368.54
Concave with Saturation Limit	0.41 s	2.0479	590.62	280.45
Concave with Saturation Limit and Downturn	3.44 s	1.9594	332.01	11754.61
S-shaped with Saturation Limit	1.05 s	1.9764	415.75	816.57
S-shaped with Saturation Limit and Downturn	1.54 s	1.9378	278.79	542.20
Gompertz Model	1.08 s	2.0242	440.67	379.59
Gaussian Model	6.96 s	2.0503	595.92	286.97

The computed quality parameter of the individual models can be found in Table 5.3 and Table 5.4.

The parameters for the fitted Gaussian model are defined in Equation 5.5 for around 8 knots and Equation 5.6 for around 14 knots. A detailed plot of both fitted models is illustrated in Figure 5.21 and Figure 5.22.

The result for the model *Concave with Saturation Limit* has infeasible high values for χ^2_{test} that probably originate from an value overflow. In Figure 5.22 this model obviously does not follow the same distribution as the data, thus it was omitted from the table.

$$f(x) = 0.8746 \cdot e^{-\frac{(x-0.6010)^2}{2\cdot 0.0729}} \tag{5.5}$$

$$f(x) = 0.8978 \cdot e^{-\frac{(x-0.6496)^2}{2\cdot 0.1101}} \tag{5.6}$$

Interpretation Compared to the model fitting of BSP and TWS it takes a lot longer to fit the models for this problem which traces back to the larger data set. The calculation

Table 5.4.: Quality values of ODR with varying models near 14 knots TWS

Model	Runtime	Durbin/Watson	χ^2_{train}	χ^2_{test}
2 degree Polynomial	0.54 s	1.9782	345.33	2177.69
3 degree Polynomial	0.79 s	1.9928	190.88	725.35
4 degree Polynomial	0.34 s	1.9896	183.17	1038.06
6 degree Polynomial	3.11 s	2.0244	164.57	450.52
Inverted Parabola	1.41 s	1.9782	345.33	2177.90
Concave with Downturn	0.49 s	1.9782	345.33	2177.79
Concave with Saturation Limit and Downturn	13.01 s	1.9782	345.38	2185.34
S-shaped with Saturation Limit	0.58 s	2.0135	377.75	530.23
S-shaped with Saturation Limit and Downturn	7.11 s	1.9544	274.56	27050.64
Gompertz Model	3.53 s	2.0151	393.95	568.76
Gaussian Model	25.65 s	2.0119	648.17	520.07

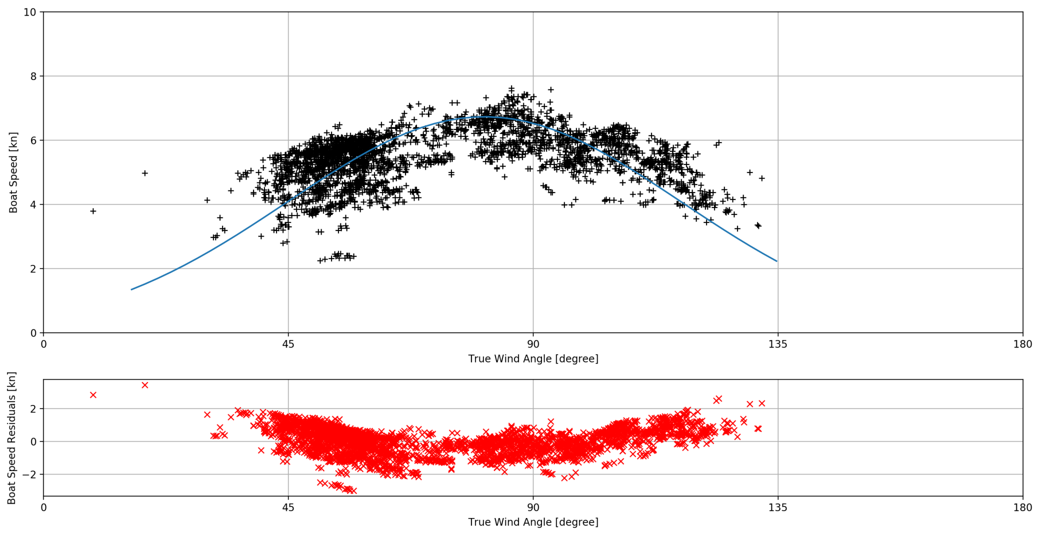


Figure 5.21.: Comparison of TWA vs. BSP from 7 to 9 knots TWS with a fitted Gaussian model and its belonging residual plot

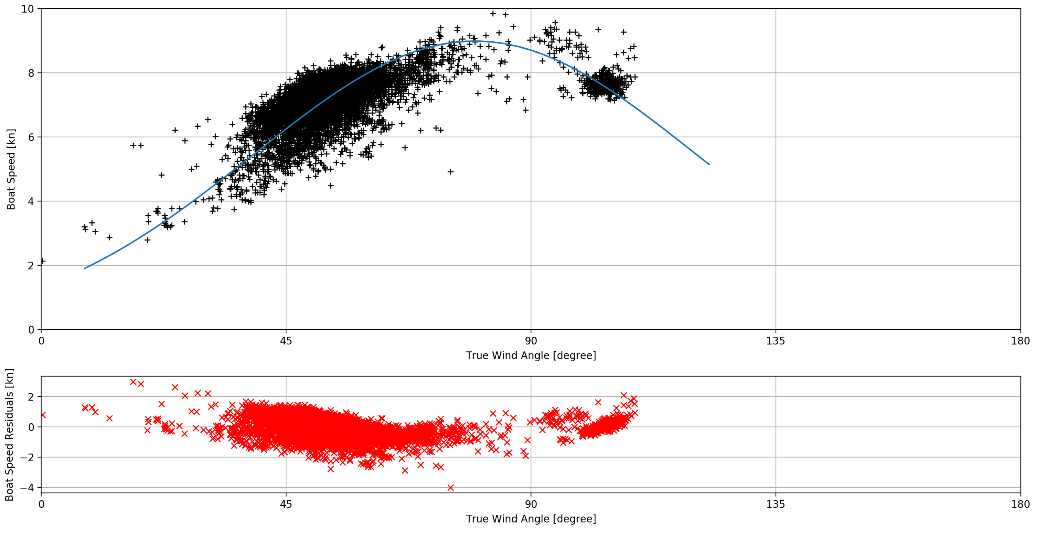


Figure 5.22.: Comparison of TWA vs. BSP from 12 to 16 knots TWS with a fitted Gaussian model and its belonging residual plot

times are about one second which is still acceptable as live-feedback.

For choosing the best model the χ^2 on the test data can be compared. In both true wind ranges the model with the lowest χ^2 on the test data is the Gaussian model. It is obvious that the best model only has one extreme value. That indicates that the sail has one optimal point and the power strictly falls off when moving away from that point. The Durbin/Watson-Test (section 3.4.2) delivers a value of 2.0269 and 1.9592 which is close to 2 and means there is no autocorrelation in this two ranges.

Both residual plots indicate that there is still a noticeable shape which means that the model does not fully represents the data. It has been found that this problem exists for all tested models (Table 5.1). It is recommended to perform further analysis to find a better model.

5.4.4. Result

The development of a higher sophisticated model of the relationship of BSP and TWA was postponed in favor of the generation of a 3-dimensional model.

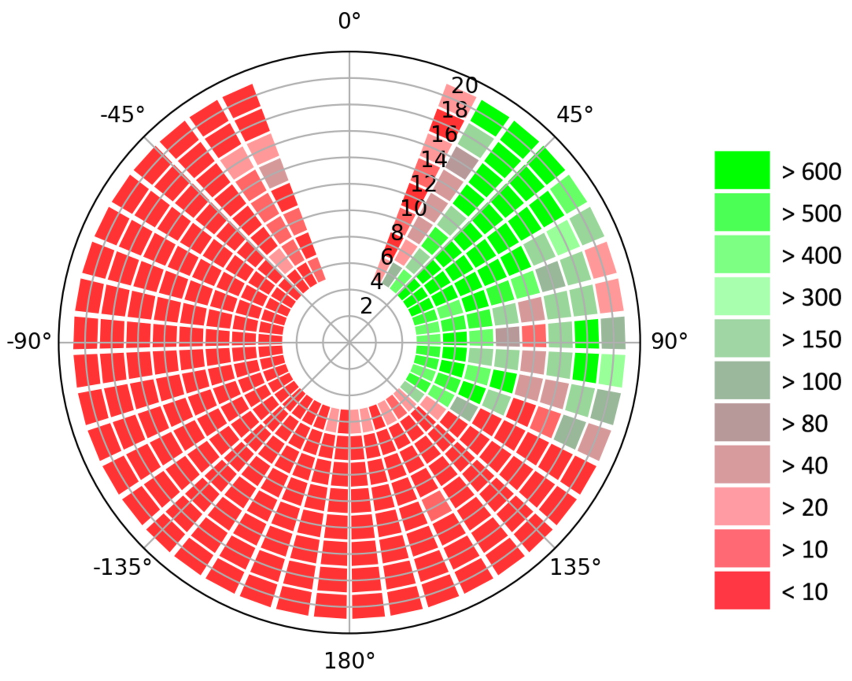


Figure 5.23.: Frequency of measured data in a polar plot.

5.5. Correlation of Boat Speed, Wind Speed and Wind Angle

Having discussed the correlation of the individual components, the following section addresses the original question of the thesis to find the influence of both components of the wind conditions on the boat speed. This is summarized in the Equation 5.7.

$$BSP = f(TWS, TWA) \tag{5.7}$$

5.5.1. Data exploration

The combination of TWS and TWA exert force on the boat that it will drive (chapter 2). Therefore these variables are independent and BSP is the dependent variable in this analysis.

Similar to the previous iterations there is a overview in which areas measured data points are available. This is illustrated in the polar diagram in Figure 5.23. TWS goes from the center outwards.

In Figure 5.24 a scatter plot of the measured data is illustrated. It is limited to the Jib on starboard bow.

5.5.2. Identification of methods, hypothesis and tests

For the extension to the third dimensions the same preconditions exist. The distance from each point to the fitted curve is calculated with the smallest geometric distance to overcome the error in all dimensions.

Two hypothesis were raised independently during the analysis. They are defined in Equation 5.8 and Equation 5.9

$$f(tws, twa) = a + b \cdot tws - c \cdot tws^2 + d \cdot twa - f\left(-e + twa\right)^2 + g \cdot twa \cdot tws \tag{5.8}$$

$$f(tws, twa) = \frac{c}{e^{a-b \cdot tws} + 1} - d \cdot tws^2 + \frac{m}{e^{n-o \cdot twa} + 1} - p \cdot twa^2 + t \cdot twa \cdot tws \tag{5.9}$$

Additionally, the two best models from the previous iterations were chosen and combined which results in the hypothesis in Equation 5.10.

$$f(tws, twa) = \frac{c}{e^{a-b \cdot tws} + 1} - d \cdot tws^2 + m \cdot e^{-\frac{(-n+twa)^2}{2 \cdot o^2}} \tag{5.10}$$

This hypothesis does not take any combined criteria into account which will be tested with an additional linear term in Equation 5.11.

$$f(tws, twa) = \frac{c}{1 + e^{a-b \cdot tws}} - d \cdot tws^2 + m \cdot e^{-\frac{(-n+twa)^2}{2 \cdot o^2}} + t \cdot twa \cdot tws \tag{5.11}$$

A model selection like in the previous iterations of the data analysis has to be made. For this multivariate nonlinear regression the best suitable quality parameter is the *chi-squared* (χ^2) as reasoned on section 3.4.2.

It is not practical to always plot the result of this fit in three dimensions including a scatter plot. Only then, the amount of data points that influence a curve at a specific position can be assumed. This is an important value of significance. Thus, in the two dimensional representation, areas with a number of data points below a certain threshold

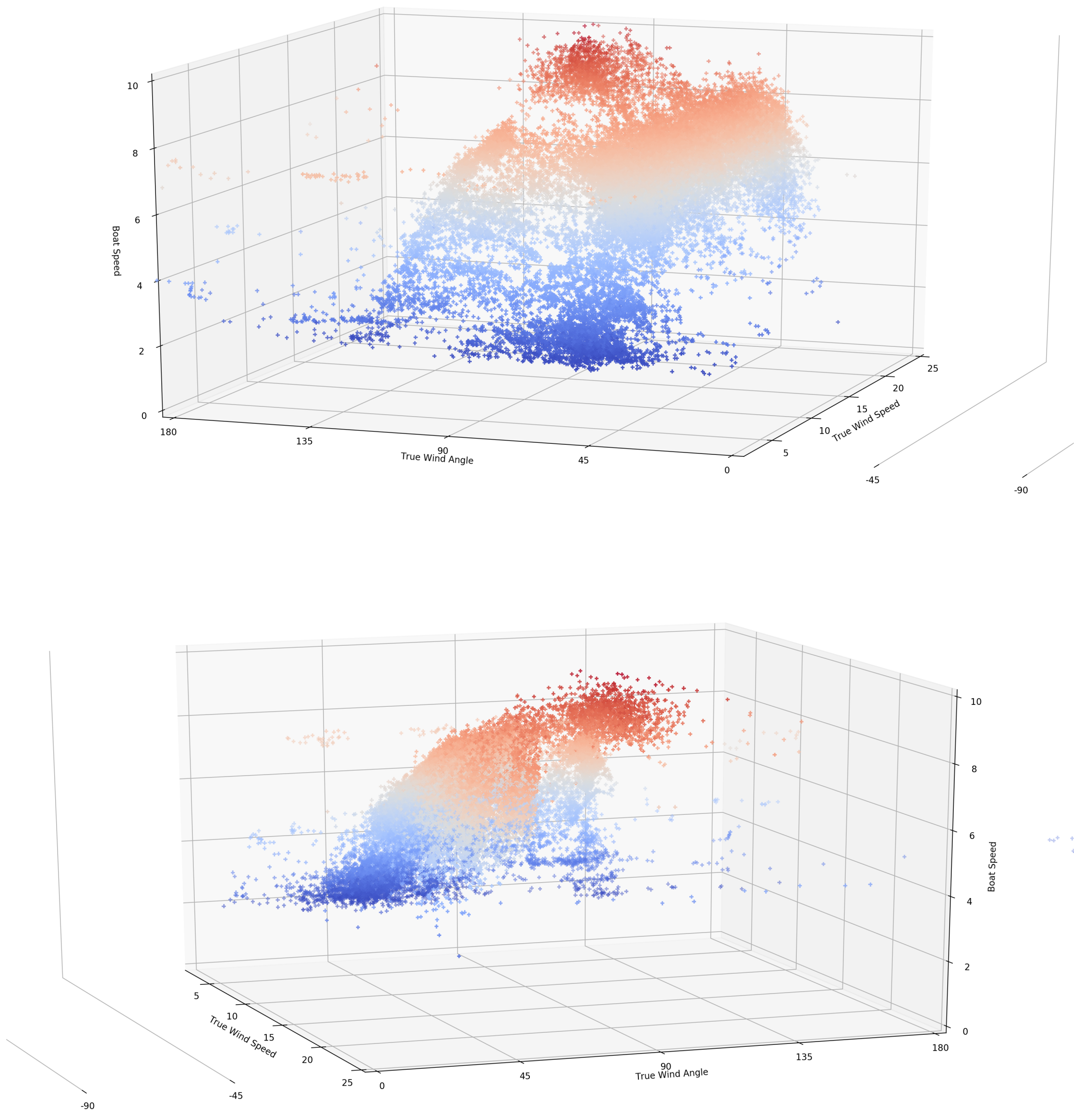


Figure 5.24.: Scatter plot of the measured data with varying perspectives.

Table 5.5.: Quality values of orthogonal distance regression of four multivariate models

Model	Runtime	$\chi^2_{training}$	χ^2_{test}
Concave with Downturn for Wind Speed and Angle (5.8)	0.18 s	47242.47	20499.41
S-shaped with Saturation Limit and Downturn for Wind Speed and Angle (5.9)	1.00 s	92678.30	39549.94
S-shaped with Sat. Limit and Downturn for Wind Speed; Gaussian model for Wind Angle (5.10)	1.20 s	38483.22	16639.00
S-shaped with Sat. Limit and Downturn for Wind Speed; Gaussian model for Wind Angle + Combination (5.11)	1.78 s	36598.24	15974.74

should be flagged or even hidden. In Figure 5.23 the areas and the corresponding histogram are illustrated. Effective frequencies below 40 contain too many insecurity for a good representation.

The official velocity prediction program, generated by the ORC, are available for this boat. A comparison of the fit to these should give another input to the validity of the hypothesis. It is not feasible that the values can be reached because the ORC calculations ignore some environmental influences (subsection 2.3.2) but the overall shape should be similar.

5.5.3. Analysis and Interpretation

With regard to the model selection the quality parameter can be found in Table 5.5. The number of data points taken into account was 91,576.

In addition to the runtime of around 2 seconds for the fit alone another 8 seconds are necessary for pre- and post-processing, thus without optimizations a *soft real-time* feedback is not feasible. It is conceivable that a continuous feedback directly on the boat every few minutes or after each lay is provided.

The models with the lowest $\chi^2_{training}$ and χ^2_{test} are the ones that were generated with the information of the previous iterations of this data analysis. The best fit is defined in

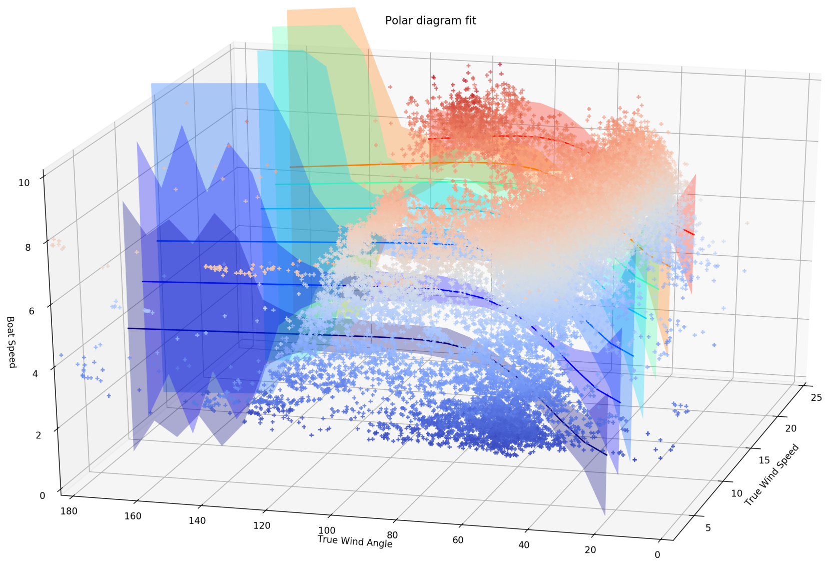


Figure 5.25.: Scatter plot of the measured data including the fitted curves of the best fitted model (Equation 5.12).

Equation 5.12. In Figure 5.25 a three dimensional plot of the fitted curves at 6, 8, 10, 12, 14, 16 and 20 knots TWS are illustrated. The areas around the fit are the indicator of the certainty of the fit at this position. It is clearly visible that with a decrease of measured data the possible result range increases.

$$\begin{aligned} f(tws, twa) = & \frac{9.8649}{1 + e^{1.6238 - 0.2765 \cdot tws}} - 0.00014 \cdot tws^2 \\ & -3.6461 \cdot e^{-\frac{(-21.1886 + twa)^2}{2 \cdot 20.6332^2}} - 0.000572 \cdot twa \cdot tws \end{aligned} \tag{5.12}$$

When delivering a 2-dimensional representation a polar plot is the preferred type. In Figure 5.26 the fit at the wind speed of 10 knots are illustrated. As reference points the values of the velocity prediction program (VPP) of this boat are added to the plot.

The fitted model defines the areas accurate where enough measurements were taken. The areas at the sides show very misleading results. Therefore in Figure 5.27 they are cut when they get inaccurate.

A split polar diagram is not useful in practice. That is the reason that another polar plot was created that combines the generated polar diagram with the VPP in continuous lines. The missing areas of the polar diagram are completed with a ratio of the VPP that is adapted from the parts of the polar diagram that contain enough measured data.

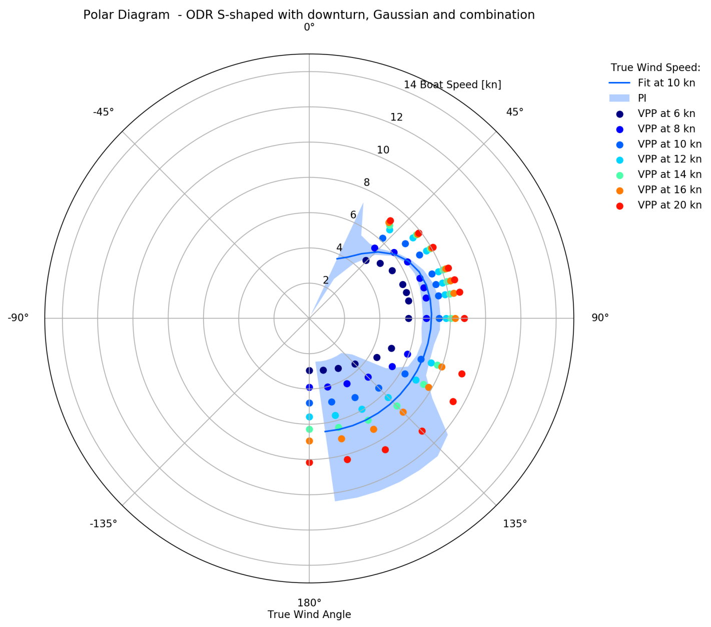


Figure 5.26.: Polar plot of the best fitted model (Equation 5.12) with the fitted line at 10 knots TWS.

5.5.4. Result

At areas with enough data points the fit seems to represent the data quite good. Better measurements with a wider range of wind angles are necessary to validate the model on the whole range of the sail. It was shown that splitting the problem into 2-dimensional subproblems assisted to find a good solution.

5.6. Correlation of Boat Speed, Wind Speed and Wind Angle in respect to Quality of Data Points

It is illustrated in Figure 5.12 and Figure 5.18 that the variance of data points is wider in some areas. That lead to the assumption that the variance of the error of the data points is not equal. In such cases, weighted regression, as described in section 3.4.1, can improve the results of the fit. For this iteration the fact that the data points are a time series is used.

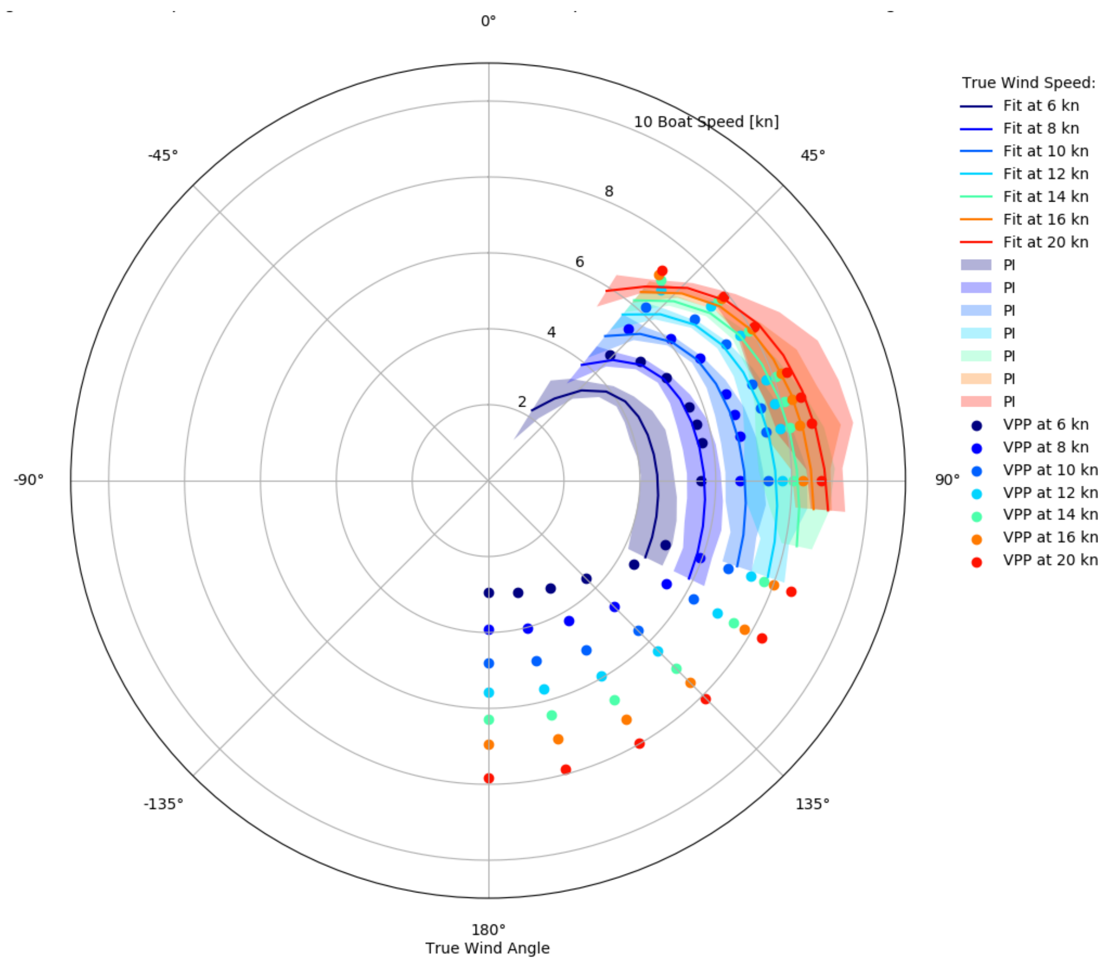


Figure 5.27.: Polar plot of the best fitted model (Equation 5.12) with accuracy above threshold.

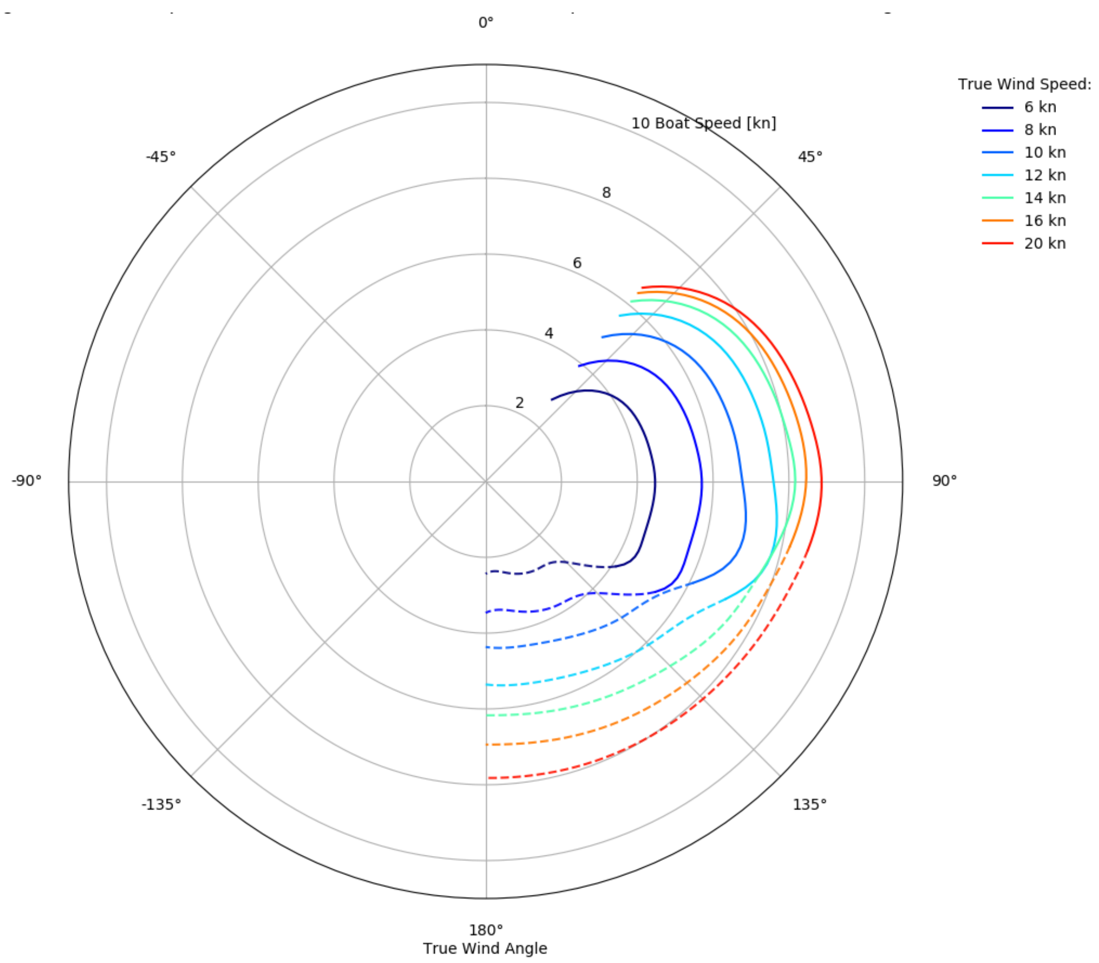
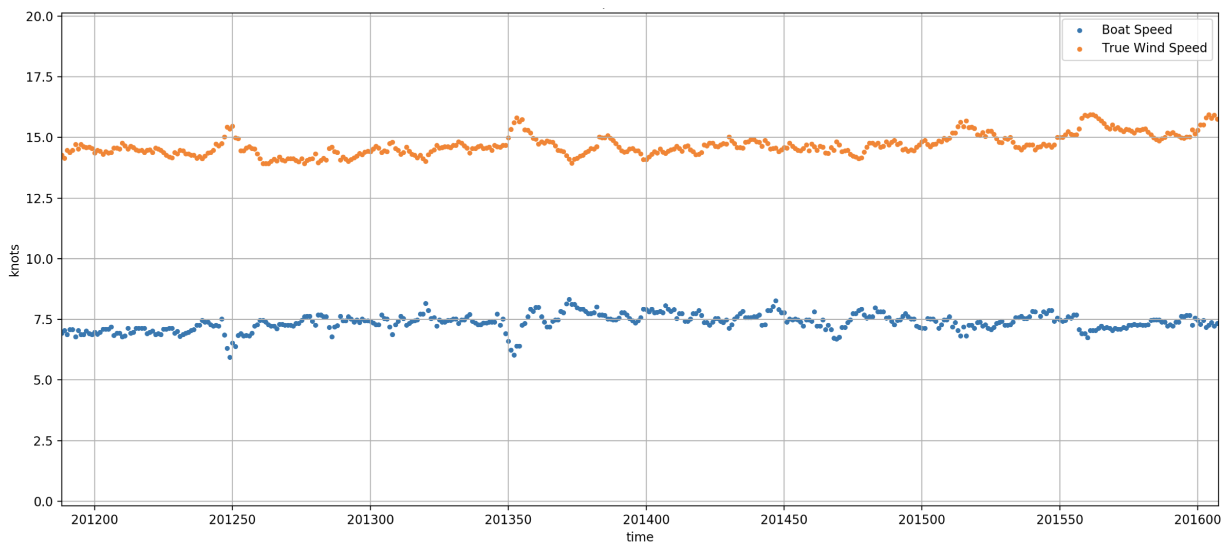


Figure 5.28.: Polar plot of the best fitted model (Equation 5.12) combined with a suitable ratio of the corresponding VPP.



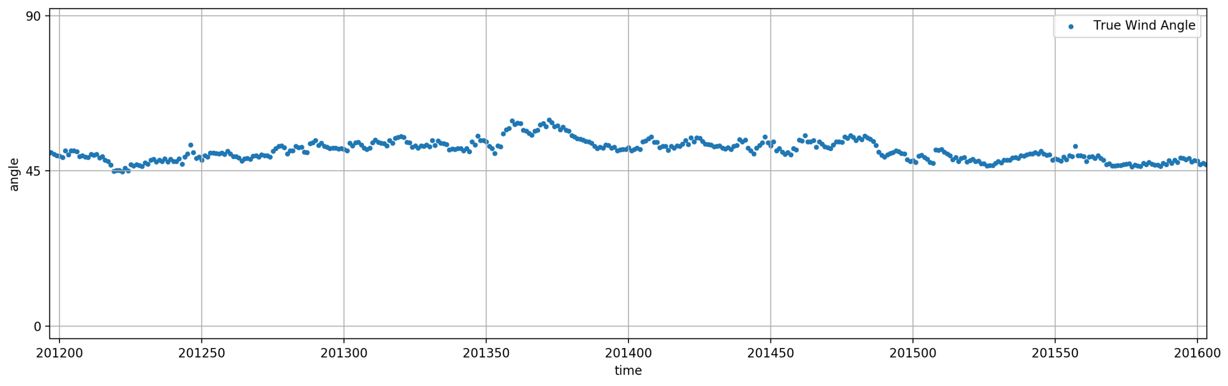


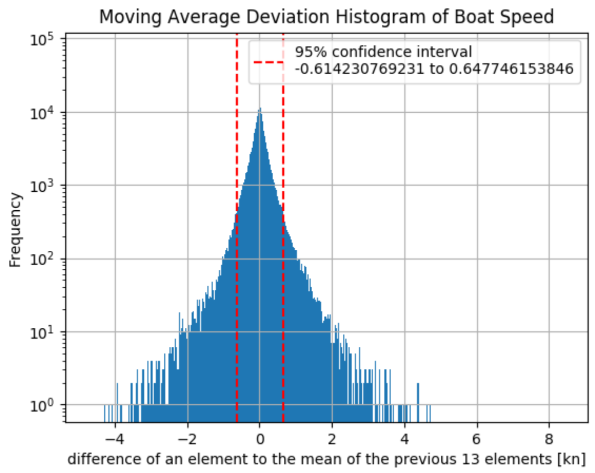
Figure 5.29.: The process of boat speed and TWS (above) and TWA (below) during a short time period of the RPC.

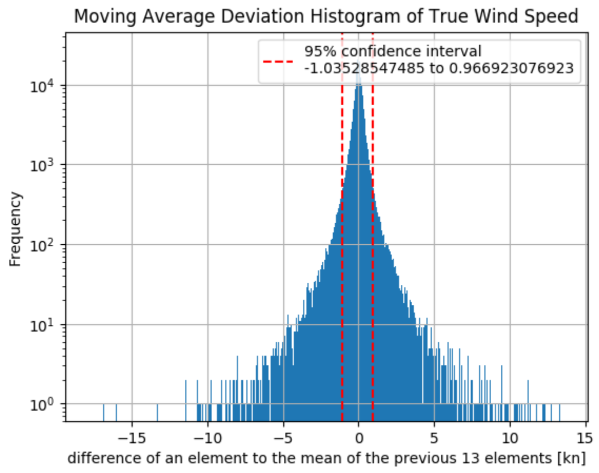
5.6.1. Data exploration

A closer look at the process of BSP and TWS is illustrated in Figure 5.29. Some areas are smooth, whereas some show sudden changes. A conclusion is discussed in the following subsection.

In Figure 5.30 the smoothness of the measured data is illustrated. Separated for each variable, it shows the deviation of the datapoints to its predecessors observed over 13 seconds.

The BSP deviates to the mean of the previous 13 seconds by up to 4 knots. When defining 5 percent of these deviations as outliers, these range drops to about 0.65 knots. This shows that an high amount of values in the time series have smooth transitions.





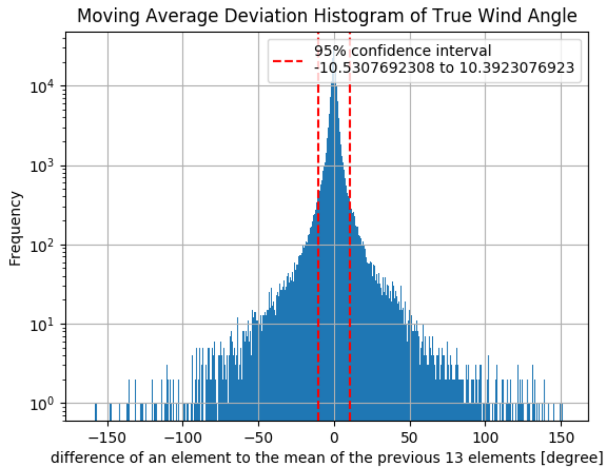


Figure 5.30.: Histograms of the difference of an element to the mean of the previous 13 seconds. The scale is logarithmic.

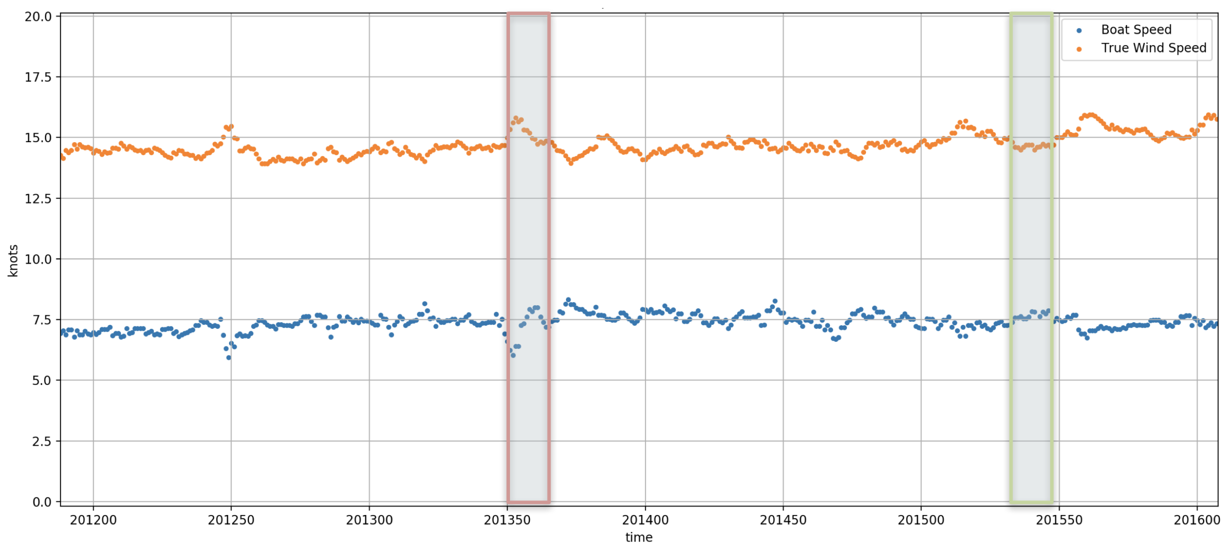


Figure 5.31.: A process diagram with stable (green) and dynamic (red) highlighted areas.

For the TWS the average deviation in the same range is around 1.0 knot and for the TWA the deviation is about 10.5 degrees.

5.6.2. Identification of methods, hypothesis and tests

Sailing is a very dynamic system. All boats react with inertia to changes of applied force. The hypothesis is that the result gets better if the stability of system during measurements is incorporated. That means that with weighted regression the measured data points during heavy changes of the system have less influence on the result.

For this calculation the data points in a time span of 13 seconds are analyzed. This value was chosen by reason of the behavior of waves in the sea.

In Figure 5.31 two of those time spans are illustrated. The green one shows smooth characteristics of the measured data. These should lead to results with low variance.

Around the time mark of 201350 seconds a change of the boat speed of over 2 knots took place. Due to the inertia of the boat the measured data in this time range probably has a bigger variance. These data points should have lesser influence on the result.

Weight calculation works as follows: For every data point the standard deviation of the previous 13 seconds is calculated. Separately for BSP, TWS and TWA. Subsequently the occurring standard deviations are analyzed for outliers. The outermost 5 percent

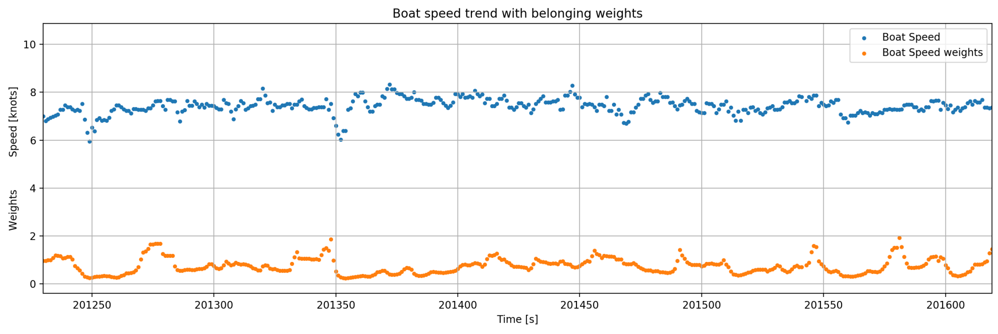


Figure 5.32.: The resulting weights of the BSP in a detailed view.

are dealt as outliers and are not included in the calculations. The weight of the others are calculated with Equation 5.13.

$$weights_i = \frac{1}{standard\_deviation(x_i, \, \dots \, , x_{i-13})^2} \tag{5.13}$$

Finally the weights are normalized using Equation 5.14.

$$normalized\_weights_i = \frac{weights_i}{\frac{1}{n}\sum_{i=1}^{n} weights_i} \tag{5.14}$$

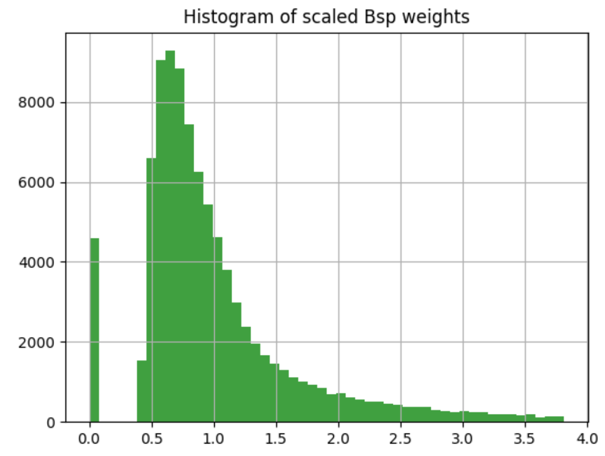
The comparison of the calculation including weights to the previous method without weights can be made by comparing the χ^2 of the test set. As discussed in section 3.4.2, this is only valid on the same training and test set and if uniform weighting is used on the test set.

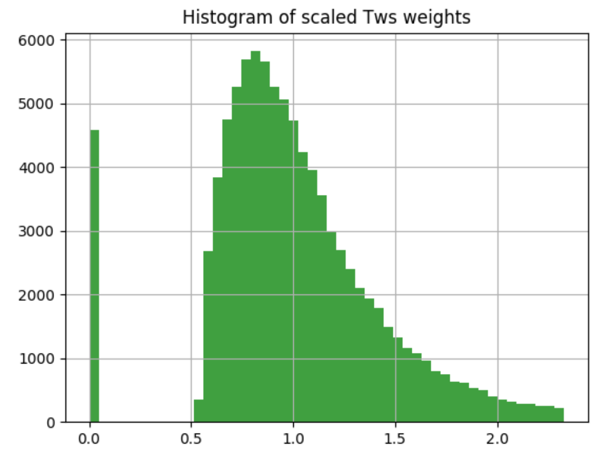
5.6.3. Analysis and Interpretation

As an example, a detailed result of the weight calculation is illustrated in Figure 5.32. It is clearly visible how the weight drops when the values of the BSP change vehemently.

The histograms of all the resulting weights that are used for the regression are illustrated in Figure 5.33. The most-left column of the histogram shows the number of outliers that are omitted for the training phase.

In Table 5.6 the fitted model including weighting of the data points is compared to the same results without weighting. It shows that the $\chi^2_{training}$, which is minimized during the regression, reaches lower values with weighted orthogonal distance regression.





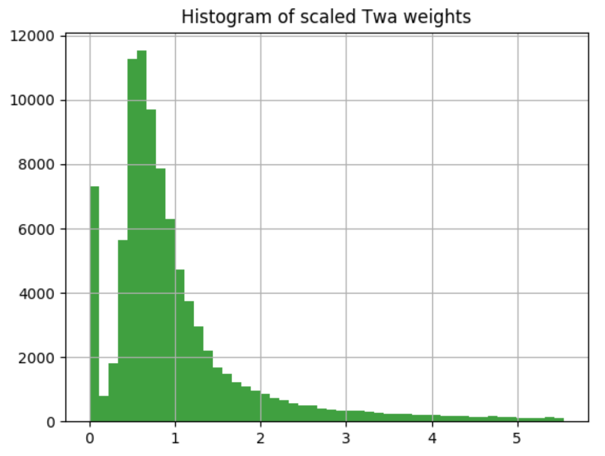


Figure 5.33.: Histograms of the resulting regression weights for BSP, TWS and TWA.

Table 5.6.: Quality values of orthogonal distance regression of four multivariate models compared to its fit with weighted data

Model	Runtime	$\chi^2_{training}$	χ^2_{test}
Concave with Downturn for Wind Speed and Angle (5.8)	0.18 s	47242.47	20499.41
with weighted data points	0.15 s	46080.42	21519.31
S-shaped with Saturation Limit and Downturn for Wind Speed and Angle (5.9)	1.00 s	92678.30	39549.94
with weighted data points	3.37 s	95739.00	61394.43
S-shaped with Sat. Limit and Downturn for Wind Speed; Gaussian model for Wind Angle (5.10)	1.20 s	38483.22	16639.00
with weighted data points	1.50 s	36679.52	17850.34
S-shaped with Sat. Limit and Downturn for Wind Speed; Gaussian model for Wind Angle + Combination (5.11)	1.78 s	36598.24	15974.74
with weighted data points	1.55 s	36474.75	17538.09

But as described in subsection 3.4.3, the column of interest is the χ^2_{test} which shows the quality of the fit with previously unseen data. These values are higher than the corresponding fitted model without weighting. Thus the weight calculation in this way is accurate, but lead to worse results when used for prediction.

The best fitted model including weighting of the data is defined in Equation 5.15.

$$\begin{aligned} f(tws, twa) = {} & \frac{1.7035}{1 + e^{0.2633 - 10.108 \cdot tws}} - 0.00023 \cdot tws^2 \\ & -1.897 \cdot e^{-\frac{(-29.678 + twa)^2}{2 \cdot -17.6511^2}} - 0.00035 \cdot twa \cdot tws \end{aligned} \tag{5.15}$$

Similar to Figure 5.26 the fit of the weighted regression is plotted with the velocity prediction programs of the ORC as reference points in Figure 5.34.

Therefore the best model remains the *S-shaped with Saturation Limit and Downturn for Wind Speed and a Gaussian model for Wind Angle plus a linear combination* which is highlighted in Table 5.6.

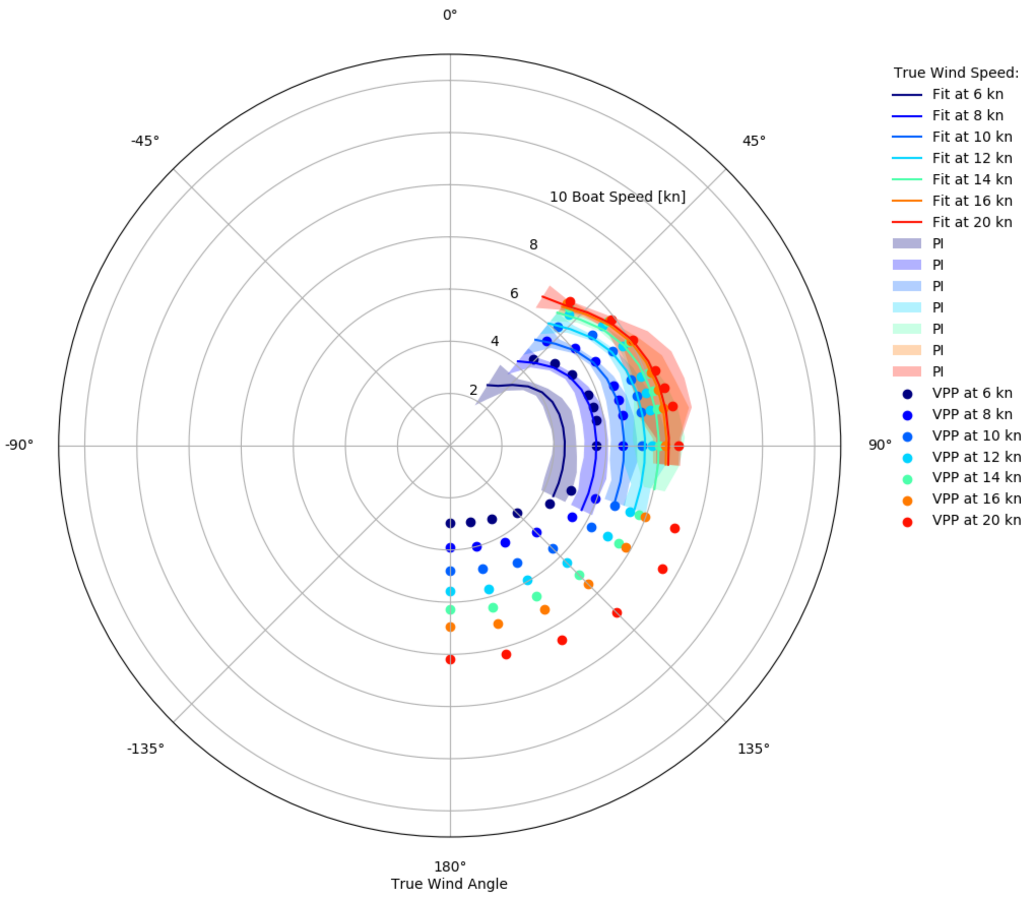


Figure 5.34.: Polar plot of the best fitted model using weighted data points (Equation 5.12) with accuracy above threshold.

5.6.4. Result

It had been shown that the weight calculation in this simple way does not lead to an improvement in the result. Maybe an approach of grading the data lead to better results.

5.7. Evaluation

Programming language The decision to use Python as a development platform was proper. In the last decade it got more and more popular in the field of data science and has a extensive documentation. No particular troubles arose and the implementation was fast.

Validity The available data do not contain information on the full range of the sails. Hence to generate a complete polar diagram is not possible. This is certainly a problem of real-world appliances, which is why a limitation of the output of the polar curves was implemented when there are not enough data available. Out of that range, the data would not be valid.

Applicability To ensure validity, the polar plot is divided in sectors. Each sector has a range of 2 knots true wind speed and 7.5 degrees true wind angle. For an sector to be valid about 100 data points in and near around the sector are necessary. That can be measured in less than two minutes sailing. A sailing boat can not sail directly against the wind, thus a feasible line in the polar diagram goes over 20 sectors. Consequently measurements for one complete line of the polar diagram take about 35 minutes. That is a feasible amount of time to invest to obtain a polar diagram.

Data selection The algorithm for the automatic selection of the test data of a specific sail works reliable. This saves a lot of time during the preparation of the data. It also lead to more accurate results than the selection of the test data from manual logbook entries. There could problems arise when handling data of a sail with different level of reef.

Modeling The correlation of TWS and BSP is good representable with the tested nonlinear models. For the connection of TWA and BSP the result is not entirely satisfying yet. Further investigation with more complex models is required. The implementation allows an extension of the analysis to any number of models with automatic comparison among each other.

Regarding the three dimensional regression, the combination of the solution of both two dimensional subtasks lead to an acceptable result. Thus the decision to divide the problem into smaller tasks is recommended. The extension of the model with a linear combination of TWS and TWA decreased the error on the fitted model of new data by 4 percent. This shows, that the best solution requires more effort than a sole composition of the subproblems.

Quality Unfortunately due to the use of nonlinear regression there is no independent calculable parameter that shows the quality of the fit (section 3.4.2). So a comparison to the VPP by the ORC can give an indicator of the quality of the fit.

It is illustrated in Figure 5.26 that for the valid areas, the shapes of the curves are quite similar. However the uniform weighted model is generally up to 1.5 knots below the velocity prediction program of that boat. The reason might be that the VPP is calculated without some influencing environmental conditions like sea or conditions of the boat and sails. This gap gets narrower to less than a half knot at higher wind speeds because the ship converges to the hull speed defined in subsection 2.3.4. The provided force by the wind increases enormous so that the environmental factors that slowdown the boat lose down.

When comparing the polar plot of the best model of weighted regression Figure 5.34 to the official velocity prediction programs like the one in Figure 2.9, it behaves rather poorly at around 40 degrees. It seems that the curve decreases not fast enough. Among the higher error on the test data, this is another sign that, given these measured data, the hypothesis without weighting of the data yield to better results.

Runtime As Table 5.5 indicates, the runtime of a fit takes only a few seconds. In addition to that there is database querying, weight calculation, determining of the fit parameters and the generation of several plots. It takes about five minutes to compare eight models at once with around 100,000 data points and all output plots. There should be room for optimization but a soft real-time feedback is not feasible.

6. Summary

The origin of this master's thesis was the master's thesis of Clemens Gutschi about weather routing (Gutschi, 2015). One of his results was, that the routing is inaccurate if the target speed was not met or over-matched. This was often the case when relying on velocity prediction program (VPP) by the Offshore Racing Congress (ORC) during his test runs. One of his outlooks was the enhancement of this input vector to the weather routing algorithm.

It was discussed that an accurate polar diagram is backing not only for weather routing but for all kinds of navigation on a sailing boat. The goal of the thesis was generating a polar diagram not from measurement of the stable parts of a boat like the VPP, but from measurements during sailing. This should lead to results that can be reproduced by the sailing crew. The research of the physical background of sailing yields that the measurement devices that are necessary for the generation are very common on sailing boats. A concept was developed that the result of this master's thesis can be applied to most of these boats. It is interconnected by the proprietary standard of the National Marine Electronics Association (NMEA). The NMEA defines a protocol to transmit data between measurement devices and displays on a boat. These transmissions are recorded and stored for the following analysis.

The analysis started with the visualization of surface splines in Matlab. This delivers visually impressive results without much effort.

To generate a complete polar diagram, data have to be recorded under varying wind conditions. Of course, those cannot be established during one measurement turn which lead to holes in the data set. It was requested to show certain results in areas that have a low number of data points. However, splines have no information of the underlying data and can show misleading values because of some dominant outliers in that area.

This circumstance lead to the domain of data analysis. Shortly at the beginning an approach with artificial neuronal networks (ANNs) was chosen. With an ANN a fit through the 3-dimensional data set should be found. The result was not satisfying and the problem was divided into smaller pieces. The 3-dimensional problem was divided into two 2-dimensional problems which were solved with regression analysis instead of an ANN. The tool for that analysis was changed to Python which has some outstanding libraries for this purpose.

Resulting from the reason that all measured values can have measurement errors, orthogonal distance regression (ODR) had to be used. During the process of model-finding, it emerged that a limitation to linear models is not accurate so that non-linear regression techniques that allow ODR were applied. That raised questions in validating the results of the fitted models. Necessary preconditions for well-known R^2 or *reduced* χ^2 does not generally hold in non-linear regression. Thus, for the validation a technique from machine learning was applied. The deviation of completely fitted models to some retained data from the same measurement is calculated. The smallest error to that previously unseen data set indicates the model that best represents the original data.

With the gained information from the lower-dimensional regressions, the bigger problem was tackled. This approach helped to preserve a feasible result. After that, a procedure was developed to improve the result by penalizing data points that might not follow the original distribution. This calculation included the fact, that the measured data are a time series and favors time ranges with stable measurements.

Detailed results can be found in section 5.7. In summary, the already mounted measurement devices deliver adequate output and the data analysis is performed in an acceptable runtime, it provides valid outcomes and requires no further knowledge of data science by the user. However, the quality of resulting polar diagrams is not quantifiable which means that the best of those generated polar diagrams is chosen but the results could still be inaccurate.

The generated polar diagram is only contingently a universally valid characteristic of the boat. If the predominant amount of the measured data is taken during unhurried sailing, the maximal performance of the boat drops. The same applies if the boat is sailed by amateur sailors.

In fact, the result polar diagram shows the performance of a particular crew on a specific boat. Thus the results can be varying, but offering new ways of applicability:

- crew compares its performance with previous accomplishments (compare Clemens Gutschi's GAP-Visualization)
- after an one-design regatta the polar diagrams of all boats can be compared and reveal at which wind condition the winner extended his lead
- as input to a weather routing program

The calculation of VPPs has some advantages that are unattainable with the generation of polar diagrams by measurements, but the tailored performance analysis can complement one another.

7. Outlook

Having discussed how to generate polar diagrams out of measurements, the final section of this thesis addresses a prospect of the next steps. A future work could be an optimization of the result by analyzing the data as time series. In the current phase, it was paid heed to not change any value of the measurements because that may affect the results negatively. Smoothing of all signals alone and to each other can be applied to the sequential data. There also might be a lag of the boat speed (BSP) when the wind conditions change resulting from an inertia effect. Techniques for the appropriate analysis of time series should be formulated and applied to this topic. The gathered results can be compared with the current hypotheses with the introduced procedures.

The resulting polar diagram currently does not spread across the whole feasible area of the sail. The reason for that issue is that the focus during the measurements was on the best velocity made good (VMG) and therefore the slower areas were avoided. Measurement data on the whole feasible area should be generated. That requires particular measurement turns to generate data of a wider range of wind conditions.

There also exists a plurality of sails that where not addressed during the analysis. These should be inspected whether the existing models are good representations. Not examined was the effect of reefing of the sails that do lead to new characteristics. A further development of models which consider this topic should be performed.

A further step is the combination of all analyzed sails of a boat into a performance polar diagram. This will show only the fastest sail for a given wind condition more conveniently. This can be used on the boat to know when to change or reef a sail.

Moreover, polar diagrams of two different time intervals should be comparable automatically. The result would express during which interval the crew accomplished a better performance. Aside from the currently averaged performance polar the implementation should be expanded to generate maximal performance polars. Besides that, when comparing both bows, an erroneous deviation of the wind angle can be identified automatically, which is a very common misconfiguration of the wind gauge.

Furthermore, in practice, it is not convenient to use a laptop to generate all this data. It needs many precious power and space on the map table. All implemented algorithms, plots and characteristic values could be ported to a handy, power-efficient device. Today's smart-phones probably have enough computing-power for such tasks. An implementation of the algorithms on mobile phones or smart-watches would enable an easy access to the generated results. In cooperation with hardware manufacturers, the generation of performance polar diagrams could also be built-in directly into the on-board devices. With either solution, the custom-fit performance polar diagrams could be delivered to the crew during sailing. This feedback can be used to adjust the trim of the sails to increase the performance.

Appendix

Appendix A.

Source Code

Listing A.1: cli.py

```
"""
Polar diagram generator.

Command line interface to access a collection of resources for the generation of polar diagrams.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import os
import logging

from datetime import datetime

import sys

from polar_diagram_generator.data_analysis import *
from polar_diagram_generator.data_selection import *
from polar_diagram_generator.exploratory_data_analysis import *
from polar_diagram_generator.workspace.workspace import Workspace

CHOOSE_COMMAND_ = \
    "\nWORKSPACE\n" \
    "open [path] - set active workspace and load db\n\tpath is relative\n" \
    "refresh (r) - refresh database of active workspace from files\n" \
    "drop (d) - drop database of active workspace\n" \
    "close (c) - close workspace\n" \
    "\n" \
    "\nCONFIGURATION\n" \
    "load_configuration [path] - set the configuration file for the data analysis\n\tpath is relative\n" \
    "select_logbook [id] - limit data for the data analysis\n" \
    "\tranges were configured manually and are available via an [id]\n" \
    "\tused frequently in the tests" \
    "select_sail_finder [angle] - limit data for the data analysis\n" \
    "\tanalyzes the data and selects ranges of a defined sail with given [angle]" \
    "select_all - removes data limitations\n" \
    "\n" \
    "\nDATA_ANALYSIS\n" \
    "trend - plot a trend of boat speed, true wind speed and true wind angle\n" \
    "trend_speed - plot a trend of true wind speed and boat speed\n" \
    "trend_wind - plot a trend of true wind angle\n" \
    "histogram - plot a histogram which shows the distribution of the data in their feasible ranges\n" \
    "boxplot - illustrate the distribution of sailed boat speeds for common ranges of true wind speed in narrow ranges of wind
        angle\n" \
    "occurrences - plot a polar heat map of the frequency of data points at narrow areas of wind angle and wind speed\n" \
    "fit [type] - fit and plot data to models\n\t* 2D_speed [angle]\n\t* 2D_angle [speed]\n\t* 3D\n" \
    "\n" \
    "\nOTHER\n" \
    "help (h) - prints this text\n" \
    "quit (q) - quit program\n"

logger = logging.getLogger('polar.cli')
```

```
def start_command_line_interface():
    configure_logging()
    db_handler = None
    workspace = Workspace()

    configuration = None

    logger.debug(CHOOSE_COMMAND_)
    number_of_commands = 0
    while True:
        number_of_commands += 1
        input_string = input(
            "Enter␣command:␣"
        )

        input_array = input_string.split()
        if len(input_array) < 1:
            continue
        command = input_array[0]
        parameters = input_array[1:len(input_array)]

        logger.debug("\nperform␣command␣%s..." % command)

        if command == "open" or command == "o" or command == "w":
            if len(parameters) is 1:
                db_handler, configuration = workspace.open_workspace(parameters[0])

            else:
                logger.debug("Usage:␣open␣[path]")

                for file in os.listdir("."):
                    if os.path.isdir("./" + file) and not file.startswith("."):
                        logger.debug("␣␣␣␣" + file)
                continue
        elif command == "close" or command == "c":
            configuration = workspace.close_workspace(configuration)
            db_handler = None
        elif command == "help" or command == "h":
            logger.debug("\n" + CHOOSE_COMMAND_)

        elif command == "quit" or command == "q":
            if workspace is not None:
                configuration = workspace.close_workspace(configuration)
            break

    # if workspace is loaded
        elif db_handler is None:
            logger.debug("no␣workspace␣loaded")
            continue

        elif command == "refresh" or command == "r":
            workspace.refresh_files()
        elif command == "drop" or command == "d":
            workspace.clear_database()

        # limits data to logbook entry
        elif command == "load_configuration":
            if len(parameters) is 1:
                configuration = workspace.configuration_from_file.override_with_config_from_file(configuration, parameters[0])
            else:
                logger.debug("Usage:␣load_configuration␣[path]\n")
                continue

        # limits data to logbook entry
        elif command == "select_logbook" or command == "l":
            configuration = logbook.perform(parameters, db_handler, configuration)

        # plot the True Wind Angle and True Wind Speed distributions
        elif command == "histogram" or command == "distribution":
            distribution.perform(db_handler, configuration)

        # plot the occurrences of data points
        elif command == "occurrences":
            occurrence_plot.perform(db_handler, configuration)

        # plot the True Wind Speed and Boat Speed trend
        elif command == "trend_speed":
            cursor = db_handler.connection.cursor()
            speed_trend.perform(cursor, configuration)
            cursor.close()

        # plot the True Wind Angle trend
        elif command == "trend_wind":
            cursor = db_handler.connection.cursor()
            wind_angle_trend.perform(cursor, configuration)
            cursor.close()

        # plot the True Wind Angle trend
        elif command == "trend":
            trend.perform(db_handler, configuration)

        # set active data range to a specific sail
        elif command == "select_sail_finder":
            select_sail_usage_string = "Usage: select_sail_finder [angle]\n\t[angle] has to be between -180 and +180\n"
            if len(parameters) is 1:
                try:
                    angle = float(parameters[0])
                    if -180 <= angle <= 180:
                        sail_finder.perform(angle, db_handler, configuration)
                except ValueError:
                    logger.debug(select_sail_usage_string)
                    continue
            else:
                logger.debug(select_sail_usage_string)
                continue

        elif command == "select_all":
            workspace.configuration_from_file.reset_selection_limitations(configuration)

        # plot the Boat Speed, True Wind Angle and True Wind Speed row difference and apply noise filtering
        elif command == "noise":
            noise_filtering.perform(db_handler, configuration)

        # plot the Boat Speed ranges for different True Wind Angle as boxplot
        elif command == "boxplot":
            cursor = db_handler.connection.cursor()
            boxplot.perform(cursor, configuration)
            cursor.close()

        elif command == "fit":
            if parameters[0] == "2D_speed" and len(parameters) is 2:  # train a 2d function in a specific True Wind Angle range
                cursor = db_handler.connection.cursor()
                fit2D_speed.perform(cursor, configuration, true_wind_angle=int(parameters[1]))
                cursor.close()
            elif parameters[0] == "2D_angle" and len(parameters) is 2:  # train a 2d function in a specific True Wind Speed
                range
                cursor = db_handler.connection.cursor()
                fit2D_angle.perform(cursor, configuration, true_wind_speed=int(parameters[1]))
                cursor.close()
            elif parameters[0] == "3D" and len(parameters) is 1:  # train a 3d function
                cursor = db_handler.connection.cursor()
                fit3D.perform(cursor, configuration)
                cursor.close()
            else:
                logger.debug("Usage: fit [type]\nChoose for [type] one of\n\t* 2D_speed [angle]\n\t* 2D_angle [speed]\n\t* 3D\n
                    \n")
                continue

        else:
            logger.debug("unknown command")
    return number_of_commands


def configure_logging():
    script_start_time = datetime.now().isoformat(timespec='minutes')
    file_name = script_start_time.replace(":", "")
    base_dir = "../logs/"
    if not os.path.isdir(base_dir):
        os.makedirs(base_dir)
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                        datefmt='%m-%d %H:%M',
                        filename=base_dir + file_name + '.log',
                        filemode='w')
    console_info = logging.StreamHandler(stream=sys.stdout)
    console_info.setLevel(logging.DEBUG)
    console_info.addFilter(LogFilter(logging.DEBUG))
    logging.getLogger('').addHandler(console_info)
    formatter = logging.Formatter('%(levelname)-8s - %(message)s')
    console_info = logging.StreamHandler(stream=sys.stdout)
    console_info.setLevel(logging.INFO)
    console_info.addFilter(LogFilter(logging.INFO))
    console_info.setFormatter(formatter)
    logging.getLogger('').addHandler(console_info)

    console_warn = logging.StreamHandler(stream=sys.stderr)
    console_warn.setLevel(logging.WARNING)
    console_warn.setFormatter(formatter)
    logging.getLogger('').addHandler(console_warn)


class LogFilter(object):
    def __init__(self, level):
        self.__level = level

    def filter(self, log_record):
        return log_record.levelno <= self.__level

if __name__ == u'__main__':
    start_command_line_interface()
```

Listing A.2: workspace/workspace.py

```
import logging
import os
from datetime import datetime

from polar_diagram_generator.importer.csvimporter import CSVImporter
from polar_diagram_generator.importer.nmeaimporter import NmeaImporter

from polar_diagram_generator.importer import import_postprocessor
from polar_diagram_generator.workspace.configuration import Configuration
from polar_diagram_generator.workspace.databasehandler import DatabaseHandler

logger = logging.getLogger('polar.workspace')


class Workspace:

    path = None
    db_handler = None
    configuration_from_file = None

    def __init__(self):
        pass

    def open_workspace(self, path):
        logger.debug("\nOpen workspace '%s'..." % path)

        if not os.path.isdir(path) or not os.access(path, os.R_OK):
            logger.error("Either path is not a folder or not readable: %s\n" % os.path.dirname(os.path.abspath(path)))
            return None, None

        files_txt = self.__list_nmea_files_in_directory(path)

        if len(files_txt) == 0:
            logger.warning("No .txt or .csv files found!\n")
            for f in os.listdir(path):
                if os.path.isdir(path + "/" + f) and not f.startswith("."):
                    logger.debug("    " + f)

            return None, None
        else:
            self.path = path

            logger.debug("Found files:")
            for f in files_txt:
                logger.debug(f)

            self.db_handler = DatabaseHandler(path, DatabaseHandler.DEFAULT_DATABASE_FILENAME)
            connection = self.db_handler.open_or_create_db()

            self.configuration_from_file = Configuration(path)
            self.configuration_from_file.write_default_config(override=False)

            configuration = self.configuration_from_file.get_default_config_from_file()
            configuration['workspace_path'] = path + '/'
            script_start_time = datetime.now().isoformat(timespec='minutes')
            output_folder_name = script_start_time.replace(":", "")
            self.__create_subfolder(output_folder_name)
            self.__create_subfolder(output_folder_name + '/plots')
            self.__create_subfolder(output_folder_name + '/models')
            self.__create_subfolder(output_folder_name + '/animation')
            configuration['output_path'] = configuration['workspace_path'] + output_folder_name + '/'

            logger.debug("\n")

            return self.db_handler, configuration

    def close_workspace(self, configuration):
        if self.db_handler is not None:
            self.path = None
            self.db_handler.close_db()

        if configuration is not None:
            configuration['workspace_path'] = ''
            configuration['output_path'] = ''
        return configuration

    def refresh_files(self):
        self.clear_database()

        # e.g. './polardata/RPC_Jasmin/2016Apr12_0.csv'
        files_txt = self.__list_nmea_files_in_directory(self.path)

        csvImporter = CSVImporter()
        nmeaImporter = NmeaImporter()
        for f in files_txt:
            nmeaImporter.import_file_to_db(self.db_handler, self.path + "/" + f)
            csvImporter.load_csv_file_to_db(self.db_handler, self.path + "/" + f)
        import_postprocessor.process(self.db_handler)

    def clear_database(self):
        self.db_handler.drop_logger_table()

    def __list_nmea_files_in_directory(self, path):
        files = os.listdir(path)
        files_txt = [f for f in files if f.endswith('.txt') or f.endswith(".csv")]
        return files_txt

    def __create_subfolder(self, folder):
        if self.path is None:
            logger.error("no_workspace_loaded")
            return
        directory_to_create = self.path + os.sep + folder + os.sep
        if not os.path.isdir(directory_to_create):
            os.makedirs(directory_to_create)
```

Listing A.3: workspace/databasehandler.py

```
import sqlite3


class DatabaseHandler:
    DEFAULT_DATABASE_FILENAME = 'default.db'

    path = None
    db_name = None
    connection = None

    def __init__(self, path, db_name=DEFAULT_DATABASE_FILENAME):
        self.path = path
        self.db_name = db_name
        pass

    def open_or_create_db(self):
        self.connection = sqlite3.connect(self.path + "/" + self.db_name)
        self.connection.row_factory = sqlite3.Row
        self.__create_default_tables()
        return self.connection

    def __create_default_tables(self):
        self.__create_logger_table_if_not_exists()
        self.__create_sails_tables_if_not_exists()
        self.__create_logbook_table_if_not_exists()

    def close_db(self):
        self.connection.close()

    def commit_transaction(self):
        self.connection.commit()

    def __create_logger_table_if_not_exists(self):
        self.connection.cursor().execute('''CREATE TABLE IF NOT EXISTS logger
                 (mark INTEGER PRIMARY KEY AUTOINCREMENT,
                  timestamp DATETIME UNIQUE,
                  latitude REAL,
                  longitude REAL,
                  bsp REAL,
                  awa REAL,
                  aws REAL,
                  twa REAL,
                  tws REAL,
                  outlier INTEGER DEFAULT 0)
                  ''')

    def __create_sails_tables_if_not_exists(self):
        connection_cursor = self.connection.cursor()
        connection_cursor.execute('''CREATE TABLE IF NOT EXISTS fore_sails
                 (id INTEGER PRIMARY KEY AUTOINCREMENT,
                  key TEXT UNIQUE,
                  name TEXT)
                  ''')
        connection_cursor.execute('''CREATE TABLE IF NOT EXISTS main_sails
                 (id INTEGER PRIMARY KEY AUTOINCREMENT,
                  key TEXT UNIQUE,
                  name TEXT)
                  ''')

    def __create_logbook_table_if_not_exists(self):
        self.connection.cursor().execute('''CREATE TABLE IF NOT EXISTS logbook
                 (id INTEGER PRIMARY KEY AUTOINCREMENT,
                  timestamp DATETIME,
                  latitude REAL,
                  longitude REAL,
                  kak INTEGER,
                  log REAL,
                  note TEXT,
                  main_sail INTEGER,
                  fore_sail INTEGER,
                  FOREIGN KEY(main_sail) REFERENCES main_sails(id),
                  FOREIGN KEY(fore_sail) REFERENCES fore_sails(id))
                  ''')

    def drop_logger_table(self):
        self.connection.cursor().execute('''DROP TABLE IF EXISTS logger''')
        self.__create_default_tables()
```

Listing A.4: workspace/configuration.py

```
import configparser
import os


class Configuration:

    def __init__(self, workspace_path):
        self.workspace_path = workspace_path
        self.config_file_name = 'config.cfg'

    def write_default_config(self, override=False):

        if override is False and os.path.isfile(self.__config_file_path()):
            return

        config = configparser.ConfigParser()

        config.add_section('Output')
        config.set('Output', 'show_title_in_plot', 'false')
        config.set('Output', 'show_plots', 'false')
        config.set('Output', 'plot_weight_histograms', 'false')
        config.set('Output', 'show_weight_histograms', 'false')
        config.set('Output', 'perform_animation_plots', 'true')

        config.add_section('SailFinder')
        config.set('SailFinder', 'deviation', '20')
        config.set('SailFinder', 'deviation_to_previous_mean', '20')

        config.add_section('DataRange')
        config.set('DataRange', 'from_timestamp', '1970-01-01_00:00:00')
        config.set('DataRange', 'to_timestamp', '2100-01-01_00:00:00')
        config.set('DataRange', 'from_wind_angle', '-180')
        config.set('DataRange', 'to_wind_angle', '360')
        config.set('DataRange', 'from_wind_speed', '2')
        config.set('DataRange', 'to_wind_speed', '25')
        config.set('DataRange', 'from_boat_speed', '2')
        config.set('DataRange', 'to_boat_speed', '20')
        config.set('DataRange', 'from_mark', '0')
        config.set('DataRange', 'to_mark', '99999999999')

        config.add_section('Models3d')
        config.set('Models3d', 'use_model_3d_tws_twa_concave', 'false')
        config.set('Models3d', 'use_model_3d_tws_twa_concave_weighted', 'false')
        config.set('Models3d', 'use_model_3d_tws_twa_s_shaped', 'false')
        config.set('Models3d', 'use_model_3d_tws_twa_s_shaped_weighted', 'false')
        config.set('Models3d', 'use_model_tws_3d_saturation_limit_twa_gauss', 'false')
        config.set('Models3d', 'use_model_tws_3d_saturation_limit_twa_gauss_weighted', 'false')
        config.set('Models3d', 'use_model_tws_3d_saturation_limit_downturn_twa_gauss', 'true')
        config.set('Models3d', 'use_model_tws_3d_saturation_limit_downturn_twa_gauss_weighted', 'false')

        with open(self.__config_file_path(), 'w') as configfile:
            config.write(configfile)

    def reset_selection_limitations(self, configuration):
        configuration['from_mark'] = 0
        configuration['to_mark'] = 99999999999
        configuration['mark_tuples'] = []
        return configuration

    def get_default_config_from_file(self):
        return self.__get_config_from_file(self.__config_file_path())

    def override_with_config_from_file(self, configuration, path):
        config = self.__get_config_from_file(path)
        for key in config.keys():
            configuration[key] = config[key]
        return configuration

    def __config_file_path(self):
        return self.workspace_path + os.sep + self.config_file_name

    def __get_config_from_file(self, path):
        config = configparser.RawConfigParser()
        config.read(path)
        configuration = {'show_title_in_plot': config.getboolean('Output', 'show_title_in_plot'),
                         'show_plots': config.getboolean('Output', 'show_plots', fallback=False),
                         'plot_weight_histograms': config.getboolean('Output', 'plot_weight_histograms', fallback=False),
                         'show_weight_histograms': config.getboolean('Output', 'show_weight_histograms', fallback=False),
                         'perform_animation_plots': config.getboolean('Output', 'perform_animation_plots'),
                         'sail_finder_deviation': config.get('SailFinder', 'deviation'),
                         'sail_finder_deviation_to_previous_mean': config.get('SailFinder', 'deviation_to_previous_mean'),
                         'from_timestamp': config.get('DataRange', 'from_timestamp'),
                         'to_timestamp': config.get('DataRange', 'to_timestamp'),
                         'from_wind_angle': config.getint('DataRange', 'from_wind_angle'),
                         'to_wind_angle': config.getint('DataRange', 'to_wind_angle'),
                         'from_wind_speed': config.getint('DataRange', 'from_wind_speed'),
                         'to_wind_speed': config.getint('DataRange', 'to_wind_speed'),
                         'from_boat_speed': config.getint('DataRange', 'from_boat_speed'),
                         'to_boat_speed': config.getint('DataRange', 'to_boat_speed'),
                         'from_mark': config.getint('DataRange', 'from_mark'),
                         'to_mark': config.getint('DataRange', 'to_mark'),
                         'use_model_3d_tws_twa_concave': config.getboolean('Models3d', 'use_model_3d_tws_twa_concave', fallback
                              =False),
                         'use_model_3d_tws_twa_concave_weighted': config.getboolean('Models3d', '
                              use_model_3d_tws_twa_concave_weighted', fallback=False),
                         'use_model_3d_tws_twa_s_shaped': config.getboolean('Models3d', 'use_model_3d_tws_twa_s_shaped',
                              fallback=False),
                         'use_model_3d_tws_twa_s_shaped_weighted': config.getboolean('Models3d', '
                              use_model_3d_tws_twa_s_shaped_weighted', fallback=False),
                         'use_model_tws_3d_saturation_limit_twa_gauss': config.getboolean('Models3d', '
                              use_model_tws_3d_saturation_limit_twa_gauss', fallback=False),
                         'use_model_tws_3d_saturation_limit_twa_gauss_weighted': config.getboolean('Models3d', '
                            use_model_tws_3d_saturation_limit_twa_gauss_weighted', fallback=False),
                        'use_model_tws_3d_saturation_limit_downturn_twa_gauss': config.getboolean('Models3d', '
                            use_model_tws_3d_saturation_limit_downturn_twa_gauss', fallback=False),
                        'use_model_tws_3d_saturation_limit_downturn_twa_gauss_weighted': config.getboolean('Models3d', '
                            use_model_tws_3d_saturation_limit_downturn_twa_gauss_weighted', fallback=False)}
        return configuration
```

Listing A.5: importer/csvimporter.py

```
"""
Imports data from CSV into the SQL database.

Intelligent detection of used separator.
Loads complete data into the database.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import csv
import logging
import os
import sqlite3

from polar_diagram_generator.workspace.databasehandler import DatabaseHandler

ALLOWED_DUPLICATE_TIMESTAMPS_PERCENTAGE = 0.005


logger = logging.getLogger('polar.importer.csv')


class CSVImporter:

    def load_csv_file_to_db(self, db_handler: DatabaseHandler, path):
        """
        Imports data from CSV into the SQL database.
        :param db_handler: reference to the database
        :param path: relative path of the file that should be imported into the database
        :return: None
        """
        if not os.path.isfile(path) or not os.access(path, os.R_OK):
            logger.error("Either file is missing or is not readable")
            return

        delimiter = CSVImporter.__guess_delimiter(path)

        with open(path) as csvfile:
            log_file_reader = csv.DictReader(csvfile, delimiter=delimiter, quotechar='"')
            cursor = db_handler.connection.cursor()
            line = 0
            imported_lines = 0
            duplicate_timestamps = 0
            for row in log_file_reader:
                line += 1
                if 'Utc' in row and 'Bsp' in row and 'Awa' in row and 'Aws' in row and 'Twa' in row and 'Tws' in row:
                    parameters = (
                        row['Utc'],
                        self.__convert_to_float_or_null(row['Bsp']),
                        self.__convert_to_float_or_null(row['Awa']),
                        self.__convert_to_float_or_null(row['Aws']),
                        self.__convert_to_float_or_null(row['Twa']),
                        self.__convert_to_float_or_null(row['Tws'])
                    )
                    try:
                        cursor.execute("INSERT INTO logger (timestamp, bsp, awa, aws, twa, tws) VALUES (?,?,?,?,?,?)",
                            parameters)
                        imported_lines += 1
                    except sqlite3.IntegrityError:
                        duplicate_timestamps += 1
                        logger.warning("%s:%s - Entry with duplicate timestamp found (%s)" % (path, line, row['Utc']))
            db_handler.commit_transaction()
            cursor.close()
            logger.debug("—— CSV–Importer ——")
            logger.debug("duplicate_timestamps:    " + str(duplicate_timestamps))
            logger.debug("stored lines:            " + str(imported_lines))
            if (imported_lines > 0 and duplicate_timestamps / imported_lines > ALLOWED_DUPLICATE_TIMESTAMPS_PERCENTAGE) or (
                imported_lines == 0 and duplicate_timestamps > 0):
                raise RuntimeWarning("There are a lot of duplicate timestamps in the imported data")
        return

    @staticmethod
    def __convert_to_float_or_null(value):
        try:
            return float(value.replace(",", "."))
        except:
            return None

    @staticmethod
    def __guess_delimiter(path):
        file_to_guess = open(path, "r")
        line = file_to_guess.readline()
        comma_count = line.count(',')
        semicolon_count = line.count(';')
        file_to_guess.close()
        if comma_count > semicolon_count:
            return ','
        else:
            return ';'
```

Listing A.6: importer/nmeaimporter.py

```
"""
Imports data from NMEA files into the SQL database.

Intelligent detection if files have a timestamp at the beginning of each line.
Identifies parallel event from the sequential format and loads complete data into the database.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging
import os.path
import sqlite3

import pynmea2

from polar_diagram_generator.importer.extended_nmea_file import ExtendedNMEAFile
from polar_diagram_generator.workspace.databasehandler import DatabaseHandler

ALLOWED_DUPLICATE_TIMESTAMPS_PERCENTAGE = 0.005

logger = logging.getLogger('polar.importer.nmea')


class NmeaImporter:
    def __init__(self):
        pass

    def import_file_to_db(self, db_handler: DatabaseHandler, file_path):
        """
        Imports data from NMEA files into the SQL database.
        :param db_handler: reference to the database
        :param path: relative path of the file that should be imported into the database
        :return: None
        """
        if not os.path.isfile(file_path) or not os.access(file_path, os.R_OK):
            logger.error("Either file is missing or is not readable")
            return

        with open(file_path) as file:

            with ExtendedNMEAFile(file) as _f:
                line = 0
                skipped_sentences = 0
                incomplete_sentences = 0
                stored_lines = 0
                unknown_sentences = 0
                duplicate_timestamps = 0
                current_datetime = None
                current_bsp = None
                current_awa = None
                current_aws = None
                current_latitude = 0.0
```

```
current_longitude = 0.0

cursor = db_handler.connection.cursor()
while True:
    try:
        sentence = _f.readline()
        line += 1
        if not hasattr(sentence, 'sentence_type'):
            unknown_sentences += 1
            continue

        if sentence.sentence_type == 'RMC' and not sentence.is_valid:
            # reset current values
            current_bsp = None
            current_awa = None
            current_aws = None
            current_datetime = None
            continue

        if sentence.sentence_type == 'RMC' and current_datetime != sentence.datetime:
            # store current values in db
            if current_datetime is not None:
                if current_bsp is None or current_awa is None or current_aws is None:
                    logger.debug("%s:%s - One or more required fields are missing at %s (bsp: %s, awa: %s, aws:
                         %s)" % (file_path, str(line), current_datetime, str(current_bsp), str(current_awa),
                         str(current_aws)))
                    incomplete_sentences += 1
                else:
                    parameters = (
                        current_datetime,
                        current_latitude,
                        current_longitude,
                        float(current_bsp),
                        float(current_awa),
                        float(current_aws)
                    )
                    try:
                        self.__add_to_database(cursor, parameters)
                        stored_lines += 1
                    except sqlite3.IntegrityError:
                        duplicate_timestamps += 1
                        logger.warning("%s:%s - Entry with duplicate RMC timestamp found %s" % (file_path, str(
                             line), current_datetime))

            # reset current values
            current_bsp = None
            current_awa = None
            current_aws = None

            current_datetime = sentence.datetime
            current_latitude = sentence.latitude
            current_longitude = sentence.longitude

        if current_datetime is None:
            logger.info("%s:%s - Sentence %s ignored (at begin of file or after GPS errors)" % (file_path, str(
                 line), sentence.sentence_type))
            skipped_sentences += 1
            continue

        # save current values
        elif sentence.sentence_type == 'VHW' and sentence.water_speed_knots is not None:
            current_bsp = sentence.water_speed_knots
        elif sentence.sentence_type == 'MWV':
            current_awa = sentence.wind_angle
            current_aws = sentence.wind_speed

        # logger.debug(sentence)
    except pynmea2.SentenceTypeError:
        logger.info("%s:%s - Sentence with unknown type found at %s" % (file_path, str(line), current_datetime)
             )
        unknown_sentences += 1
    except TypeError:
        logger.info("%s:%s - TypeError occurred after %s (e.g. missing GPS datetime)" % (file_path, str(line),
             current_datetime))
        unknown_sentences += 1
    except pynmea2.ParseError:
        break  # eof?

db_handler.commit_transaction()
cursor.close()
```

```
            logger.debug("----- " + file.name)
            logger.debug("----- NMEA-Importer -----")
            logger.debug("skipped_sentences:        " + str(skipped_sentences))
            logger.debug("duplicate_timestamps:     " + str(duplicate_timestamps))
            logger.debug("incomplete_sentences:     " + str(incomplete_sentences))
            logger.debug("stored_lines:             " + str(stored_lines))
            logger.debug("unknown_sentences:        " + str(unknown_sentences))
            if (stored_lines > 0 and duplicate_timestamps / stored_lines > ALLOWED_DUPLICATE_TIMESTAMPS_PERCENTAGE) or (
                    stored_lines == 0 and duplicate_timestamps > 0):
                raise RuntimeWarning("There are a lot of duplicate timestamps in the imported data")
        return skipped_sentences, incomplete_sentences, stored_lines, unknown_sentences

    def __add_to_database(self, cursor, parameters):
        cursor.execute("INSERT INTO logger (timestamp, latitude, longitude, bsp, awa, aws) VALUES (?,?,?,?,?,?)",
                       parameters)
```

Listing A.7: importer/import˙postprocessor.py

```
"""
Normalize and complete database after an import.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging

from polar_diagram_generator.converter import wind_converter
from polar_diagram_generator.workspace.databasehandler import DatabaseHandler

logger = logging.getLogger('polar.importer.postprocessing')


def process(db_handler: DatabaseHandler):
    """
    Normalize and complete database after an import:
    * removes rows with negative BSP
    * calculates apparent wind conditions
    * calculates true wind conditions
    * converts wind angle
    :param db_handler: reference to the database
    """
    __remove_rows_with_negative_bsp(db=db_handler)
    __calculate_and_set_apparent_wind_values(db=db_handler)
    __calculate_and_set_true_wind_values(db=db_handler)
    __convert_wind_angle_to_180_to_180(db=db_handler)


def __remove_rows_with_negative_bsp(db):
    cursor = db.connection.cursor()
    data = cursor.execute('DELETE FROM logger WHERE bsp < 0')
    logger.info("rows with invalid bsp deleted: " + str(data.rowcount))
    db.commit_transaction()
    cursor.close()
    return


def __load_all_wind_angle_values(cursor):
    cursor.execute('SELECT mark, awa, twa FROM logger'
                   ' WHERE awa IS NOT NULL'
                   ' AND twa IS NOT NULL')
    return cursor.fetchall()


def __convert_wind_angle_to_180_to_180(db):
    cursor = db.connection.cursor()
    wind_angle_values = __load_all_wind_angle_values(cursor=cursor)
    adjusted_lines = 0
    skipped_lines = 0
    faulty_lines = 0
    for wind_angle_value in wind_angle_values:
        mark = wind_angle_value[0]
        awa = wind_angle_value[1]
        twa = wind_angle_value[2]

        try:
            new_awa = wind_converter.wind_angle_convert_to_range_180_to_180(angle=awa)
            new_twa = wind_converter.wind_angle_convert_to_range_180_to_180(angle=twa)
            if awa == new_awa and twa == new_twa:
                skipped_lines += 1
                continue

            parameters = (new_awa, new_twa, mark)
            cursor.execute("UPDATE logger SET awa = (?), twa = (?) WHERE mark IS ?", parameters)
            adjusted_lines += 1
        except:
            faulty_lines += 1

    logger.debug("converted wind angle range for #rows: " + str(adjusted_lines) + " (" + str(skipped_lines) + " skipped, " +
         str(faulty_lines) + " faulty)")
    db.commit_transaction()
    cursor.close()


def __load_unset_apparent_wind_values(cursor):
    cursor.execute('SELECT mark, bsp, twa, tws FROM logger'
                   ' WHERE awa IS NULL'
                   ' AND aws IS NULL')
    return cursor.fetchall()


def __calculate_and_set_apparent_wind_values(db):
    cursor = db.connection.cursor()
    entries_with_missing_apparent_wind = __load_unset_apparent_wind_values(cursor=cursor)
    adjusted_lines = 0
    faulty_lines = 0
    for entry_with_missing_apparent_wind in entries_with_missing_apparent_wind:
        mark = entry_with_missing_apparent_wind[0]
        bsp = entry_with_missing_apparent_wind[1]
        twa = entry_with_missing_apparent_wind[2]
        tws = entry_with_missing_apparent_wind[3]

        try:
            awa, aws = wind_converter.true_to_apparent_wind(twa, tws, bsp)

            parameters = (awa, aws, mark)
            cursor.execute("UPDATE logger SET awa = (?), aws = (?) WHERE mark IS ?", parameters)
            adjusted_lines += 1
        except:
            faulty_lines += 1

    logger.debug("calculated apparent wind for #rows: " + str(adjusted_lines) + " (" + str(faulty_lines) + " skipped)")
    db.commit_transaction()
    cursor.close()


def __load_unset_true_wind_values(cursor):
    cursor.execute('SELECT mark, bsp, awa, aws FROM logger'
                   ' WHERE twa IS NULL'
                   ' AND tws IS NULL')
    return cursor.fetchall()


def __calculate_and_set_true_wind_values(db):
    cursor = db.connection.cursor()
    entries_with_missing_true_wind = __load_unset_true_wind_values(cursor=cursor)
    adjusted_lines = 0
    faulty_lines = 0
    for entry_with_missing_true_wind in entries_with_missing_true_wind:
        mark = entry_with_missing_true_wind[0]
        bsp = entry_with_missing_true_wind[1]
        awa = entry_with_missing_true_wind[2]
        aws = entry_with_missing_true_wind[3]

        try:
            twa, tws = wind_converter.apparent_to_true_wind(awa, aws, bsp)

            parameters = (twa, tws, mark)
            cursor.execute("UPDATE logger SET twa = (?), tws = (?) WHERE mark IS ?", parameters)
            adjusted_lines += 1
        except:
            faulty_lines += 1

    logger.debug("calculated true wind for #rows: " + str(adjusted_lines) + " (" + str(faulty_lines) + " skipped)")
    db.commit_transaction()
    cursor.close()
```

Listing A.8: converter/wind˙converter.py

```
""" This file contains converters of wind types """

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0


from math import radians, degrees, isnan

import numpy
from numpy.ma import arccos, cos, sqrt


def apparent_to_true_wind(awa, aws, bsp):
    """
    Converts apparent wind conditions to true wind conditions
    :param awa: apparent wind angle
    :param aws: apparent wind speed
    :param bsp: boat speed
    :return: true wind angle, true wind speed
    """
    if isnan(awa) or isnan(aws) or isnan(bsp):
        raise ValueError('parameters have to be numbers')
    if bsp < 0:
        raise ValueError('Boat Speed has to be greater 0')

    awa_is_negative = awa < 0
    awa_is_above_180 = awa > 180

    awa_rad = radians(awa)
    tws = sqrt(pow(aws, 2) + pow(bsp, 2) - 2 * aws * bsp * cos(awa_rad))
    twa = arccos((aws * cos(awa_rad) - bsp) / tws)

    if awa_is_negative:
        result_twa = 0 - degrees(twa)
    elif awa_is_above_180:
        result_twa = 360 - degrees(twa)
    else:
        result_twa = degrees(twa)
    return round(result_twa, 1), tws


def true_to_apparent_wind(twa, tws, bsp):
    """
    Converts true wind conditions to apparent wind conditions
    :param twa: true wind angle
    :param tws: true wind speed
    :param bsp: boat speed
    :return: apparent wind angle, apparent wind speed
    """
    if isnan(twa) or isnan(tws) or isnan(bsp):
        raise ValueError('parameters have to be numbers')
    if bsp < 0:
        raise ValueError('Boat Speed has to be greater 0')

    twa_is_negative = twa < 0
    twa_is_above_180 = twa > 180

    twa_rad = radians(twa)
    aws = sqrt(pow(tws, 2) + pow(bsp, 2) + 2 * tws * bsp * cos(twa_rad))
    awa = arccos((tws * cos(twa_rad) + bsp) / aws)

    if twa_is_negative:
        result_awa = 0 - degrees(awa)
    elif twa_is_above_180:
        result_awa = 360 - degrees(awa)
    else:
        result_awa = degrees(awa)

    return round(result_awa, 1), aws


def wind_angle_cart_2_pol(angle):
    """
    Converts cartesian coordinates to polar coordinates
    :param angle: angle in cartesian coordinates
    :return: polar coordinates
    """
    angle %= 360
    return ((2 * numpy.pi) / 360) * angle


def wind_angle_pol_2_cart(angle):
    """
    Converts polar coordinates to cartesian coordinates
    :param angle: angle in polar coordinates
    :return: cartesian coordinates
    """
    angle %= 2 * numpy.pi
    return (360 / (2 * numpy.pi)) * angle


def wind_angle_convert_to_range_0_to_360(angle):
    """
    Converts wind angle from range -180 to +180 degrees to range 0 to 360 degrees
    :param angle: wind angle from -180 to +180 degrees
    :return: wind angle from 0 to 360 degrees
    """
    if isnan(angle):
        raise ValueError('parameters have to be numbers')
    angle %= 360
    angle_is_negative = angle < 0

    if angle_is_negative:
        result = 360 - angle
    else:
        result = angle

    return result


def wind_angle_convert_to_range_180_to_180_list(angles, tuple_index=0):
    """
    Converts a list of wind angles from range 0 to 360 degrees to range -180 to +180 degrees
    :param angles: list of wind angles from 0 to 360 degrees
    :param tuple_index: if angles contains tuples, selects the correct index
    :return: wind angle from -180 to +180 degrees
    """
    converted_angles = []
    for angle in angles:
        angle_ = angle[tuple_index]
        if None == angle_ or isnan(angle_):
            converted_angles.append((None))
        else:
            converted_angles.append(wind_angle_convert_to_range_180_to_180(angle_))
    return converted_angles


def wind_angle_convert_to_range_180_to_180(angle):
    """
    Converts wind angle from range 0 to 360 degrees to range -180 to +180 degrees
    :param angle: wind angle from 0 to 360 degrees
    :return: wind angle from -180 to +180 degrees
    """
    if isnan(angle):
        raise ValueError('parameters have to be numbers')
    angle %= 360
    angle_is_above_180 = angle > 180

    if angle_is_above_180:
        result = -360 + angle
    else:
        result = angle

    return result
```

Listing A.9: data˙selection/logbook.py

```
"""
Adjust the configuration to use subsets of the data.

These values were manually analyzed and are only for the following data-sets:
- Round Palagruza Cannonball 2016
- Austria One 2016

"""
```

```python
# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging

logger = logging.getLogger('polar.data_selection.logbook')


def perform(parameters, db_handler, configuration):
    """
    Adjust the configuration to use subsets of the data.
    :param parameters: additional parameters from the command line. First parameter should be an [id] e.g. rpc_all
    :param db_handler: database handler with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    :return: an updated configuration array
    """
    if parameters is None or len(parameters) != 1:
        logger.debug("Usage: logbook <id>")
        return configuration

    first_parameter = parameters[0]
    if first_parameter == "rpc_all":
        configuration['from_mark'] = 98513  # manually analyzed the start of RPC at 12:00
        configuration['to_mark'] = 394110  # manually analyzed
        configuration['from_timestamp'] = '2016-04-11 12:00:00'
        configuration['to_timestamp'] = '2016-04-14 23:51:49'
        configuration['to_wind_speed'] = 30
        configuration['to_boat_speed'] = 10
    elif first_parameter == "rpc_1":  # manually analyzed; only one lay
        configuration['from_mark'] = 249739
        configuration['to_mark'] = 282131
        configuration['to_wind_speed'] = 30
        configuration['to_boat_speed'] = 10
    elif first_parameter == "rpc_2":  # manually analyzed; only one sail
        configuration['from_mark'] = 98513
        configuration['to_mark'] = 303900
        configuration['to_wind_speed'] = 30
        configuration['to_boat_speed'] = 10
    elif first_parameter == "rpc_spi":
        configuration['from_mark'] = 303900
        configuration['to_mark'] = 339200
        configuration['to_wind_speed'] = 30
        configuration['to_boat_speed'] = 10

    elif first_parameter == "a1_1":
        configuration['from_timestamp'] = '2016-07-27 06:35:00'
        configuration['to_timestamp'] = '2016-07-27 08:30:00'
        configuration['from_mark'], configuration['to_mark'] = __mark_from_start_end_date(
            db_handler.get_cursor(),
            configuration['from_timestamp'],
            configuration['to_timestamp'])
    elif first_parameter == "a1_2":
        configuration['from_timestamp'] = '2016-07-25 16:49:02'
        configuration['to_timestamp'] = '2016-07-26 05:18:13'
        configuration['from_mark'], configuration['to_mark'] = __mark_from_start_end_date(
            db_handler.get_cursor(),
            configuration['from_timestamp'],
            configuration['to_timestamp'])
    elif first_parameter == "a1_3":
        configuration['from_timestamp'] = '2016-07-24 05:45:00'
        configuration['to_timestamp'] = '2016-07-24 06:45:00'
        configuration['from_mark'], configuration['to_mark'] = __mark_from_start_end_date(
            db_handler.get_cursor(),
            configuration['from_timestamp'],
            configuration['to_timestamp'])
    elif first_parameter == "a1_4":
        configuration['from_timestamp'] = '2016-07-28 02:04:32'
        configuration['to_timestamp'] = '2016-07-28 08:38:51'
        configuration['from_mark'], configuration['to_mark'] = __mark_from_start_end_date(
            db_handler.get_cursor(),
            configuration['from_timestamp'],
            configuration['to_timestamp'])
    else:
        logger.error("logbook entry with id '%s' not found!\n" % first_parameter)

    return configuration


def __mark_from_start_end_date(cursor, from_timestamp, to_timestamp):
    parameters = (from_timestamp, to_timestamp)
    cursor.execute('SELECT mark FROM logger '
                   ' WHERE ((timestamp > ? AND timestamp <= ?))'
                   , parameters)
    marks_in_range = cursor.fetchall()
    from_mark = marks_in_range[0][0]
    to_mark = marks_in_range[len(marks_in_range) -1][0]
    return from_mark, to_mark
```

Listing A.10: data˙selection/sail˙finder.py

```
"""
Identifies all data points of a sail without prior labelling.

Relies on the common way of sailing of using a stable wind angle.
Returns all ranges where a sail with a defined wind angle was hoisted.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import matplotlib.pyplot as plt
import numpy

import polar_diagram_generator.converter.wind_converter as wind_converter
from polar_diagram_generator.exploratory_data_analysis import wind_angle_trend
from polar_diagram_generator.workspace.databasehandler import DatabaseHandler


def __load_values(cursor, configuration):
    parameters = (configuration['from_mark'], configuration['to_mark'])
    cursor.execute('SELECT mark, bsp, twa, tws FROM logger'
                   ' WHERE bsp IS NOT NULL'
                   ' AND twa IS NOT NULL'
                   ' AND tws IS NOT NULL'
                   ' AND ((mark > ? AND mark <= ?) OR mark IS NULL)'
                   , parameters)
    val = cursor.fetchall()
    return val


def plot(differences_array, upper, lower, title, unit, kernel_size, file_path, show_title=True):
    plt.figure()
    if show_title:
        plt.title("Moving Average Deviation Histogram of " + title)
    bins = numpy.linspace(-1, 1, 0.1)
    plt.hist(differences_array, bins=1000, log=True)
    plt.xlabel("difference of an element to the mean of the previous " + str(kernel_size) + " elements [" + unit + "]")
    plt.ylabel("Frequency")
    label = "99% confidence interval\n" + str(lower) + " to " + str(upper)
    lower_line = plt.axvline(lower, label=label, color="r", ls="--")
    plt.axvline(upper, color="r", ls="--")
    plt.legend(handles=[lower_line])
    plt.grid(True)
    plt.draw()
    plt.savefig(file_path)


def perform(angle, db_handler: DatabaseHandler, configuration):
    """
    Identifies all data points of a sail without prior labelling.
    :param angle: True wind angle with the estimated best VMG of the sail
    :param db_handler: database handler with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    __print_twa(angle, db_handler, configuration)


def __print_twa(angle, db_handler: DatabaseHandler, configuration):
    target_angle = float(angle)  # range -180 to +180
    deviation = float(configuration['sail_finder_deviation'])
    deviation_to_previous_mean = float(configuration['sail_finder_deviation_to_previous_mean'])

    start_end_tuples = []
    cursor = db_handler.connection.cursor()
    count_result = __load_values(cursor, configuration)

    window_size = 20
    range_active = False
    start_mark = 0
    previous_mean = float(99999)

    for index in range(window_size, len(count_result), window_size):
        if index >= window_size:
            sum_twa_0_to_360 = 0
            sum_twa_180_to_180 = 0
            for i in range(-window_size+1, 1):
                sum_twa_180_to_180 += wind_converter.wind_angle_convert_to_range_180_to_180(count_result[index + i][2])

            mean_of_window = sum_twa_180_to_180 / window_size

            is_in_range_of_previous_mean = previous_mean - deviation_to_previous_mean < mean_of_window < previous_mean +
                 deviation_to_previous_mean
            previous_mean = mean_of_window
            index_of_window_start = index - window_size + 1
            mark_of_window_start = count_result[index_of_window_start][0]
            mark_of_window_end = count_result[index][0]

            if mark_of_window_end - mark_of_window_start > 2 * window_size:
                range_active = False
                end_mark = mark_of_window_start
                start_end_tuples.append((start_mark, end_mark))
                continue

            if target_angle - deviation < mean_of_window < target_angle + deviation or (range_active and
                 is_in_range_of_previous_mean):
                if range_active:
                    continue
                else:
                    range_active = True
                    start_mark = count_result[index_of_window_start][0]
            else:
                if range_active:
                    range_active = False
                    end_mark = mark_of_window_end
                    start_end_tuples.append((start_mark, end_mark))

    mark, twa = wind_angle_trend.wind_angle_values(cursor, configuration)
    twa_180 = wind_converter.wind_angle_convert_to_range_180_to_180_list(twa)
    if configuration['show_title_in_plot']:
        title = "Scatter Diagram of True Wind Angle"
    else:
        title = None
    wind_angle_trend.plot(mark, twa_180, title, configuration['output_path'] + "plots/sail_selection_angle_trend.png")

    for start, end in start_end_tuples:
        wind_angle_trend.add_active_logbook_indicator(start, end)

    # outliers = []
    # cursor.execute('UPDATE logger SET outlier = 0 WHERE outlier IS 1')
    # for difference in differences:
    #     if not (lower_bsp < difference[1] < upper_bsp and lower_twa < difference[2] < upper_twa and lower_tws < difference[4]
         < upper_tws):
    #         cursor.execute('UPDATE logger SET outlier = 1 WHERE mark IS ?', (difference[0],))
    #         outliers.append(difference[0])
    #
    # db_handler.commit_transaction()

    if configuration['show_plots'] is True:
        plt.show()
    else:
        plt.draw()

    if len(start_end_tuples) > 0:
        plt.savefig(configuration['output_path'] + "plots/sail_selection_angle_trend_with_selected_sail.png")

    configuration["mark_tuples"] = start_end_tuples
    cursor.close()
```

Listing A.11: exploratory_data_analysis/boxplot.py

```
"""
Illustrates multiple boxplots which boat speeds are common for some true wind speeds and angles.

Performs exploratory data analysis of the data.
Loads data from the database as defined and generates several plots.
All of them contain multiple boxplots for narrow wind angles
```

```
which illustrate the distribution of sailed boat speeds for common ranges of true wind speed in that ranges.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging

import matplotlib.pylab as plt
import numpy

logger = logging.getLogger('polar.exploratory_data_analysis.boxplot')


def perform(cursor, configuration):
    """
    Illustrates multiple boxplots which boat speeds are common for some true wind speeds and angles.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    for ws in range(configuration['from_wind_speed'], configuration['to_wind_speed'] + 1, 2):
        split_bsp = __load_bsp_histogram_values(cursor, 5, ws)
        __extract_std(split_bsp)
        __plot(split_bsp, ws, configuration)
        __plot_polar(split_bsp, ws, configuration)
    plt.show()


def __load_bsp_histogram_values(cursor, step_size, tws_area):
    bsp = []
    for i in range(0, 180, step_size):
        parameters = (i, i + step_size, tws_area, tws_area)
        cursor.execute('SELECT bsp FROM logger'
                       ' WHERE bsp IS NOT NULL'
                       ' AND twa IS NOT NULL'
                       ' AND tws IS NOT NULL'
                       ' AND abs(twa) >= ?'
                       ' AND abs(twa) < ?'
                       ' AND tws >= ?-1'
                       ' AND tws < ?+1'
                       , parameters)
        val = cursor.fetchall()
        bsp.append(val)
    return bsp


def __plot(bsp, wind_speed, configuration):
    plt.figure()
    plt.title("Boat Speed Boxplot " + str(wind_speed) + "kn")
    plt.ylim(configuration['from_boat_speed'], configuration['to_boat_speed'])
    plt.boxplot(bsp, 1, 'gD')
    plt.xlabel("True Wind Angle [/20]")
    plt.ylabel("Boat Speed")
    plt.grid(True)
    plt.draw()
    plt.savefig(configuration['output_path'] + "plots/boxplot_bsp_twa_" + str(wind_speed) + ".png")


def __plot_polar(bsp, wind_speed, configuration):
    plt.figure()
    ax = plt.subplot(111, projection='polar')
    ax.set_title("Boat Speed Boxplot " + str(wind_speed) + "kn")
    ax.boxplot(bsp, 1, 'gD')
    plt.draw()
    plt.savefig(configuration['output_path'] + "plots/boxplot_polar_bsp_twa_" + str(wind_speed) + ".png")


def __extract_std(list):
    for element in list:
        logger.debug(len(element), end='\t')
        logger.debug(numpy.std(numpy.asarray(element)), end='\t')
        logger.debug(numpy.std(numpy.asarray(element)) / len(element))
    logger.debug("")
```

Listing A.12: exploratory_data_analysis/distribution.py

```
"""
Illustrates separate histograms of true wind speed, true wind angle and boat speed.
```

```
Performs exploratory data analysis of the data.
Loads data from the database as requested and generates normal and polar plots.
This shows the distribution of the data in their feasible ranges.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging

import matplotlib.pylab as plt
import numpy

from polar_diagram_generator.converter import wind_converter
from polar_diagram_generator.data_analysis import fit3D
from polar_diagram_generator.util.strings import TWA_TICK_LABELS
from polar_diagram_generator.workspace.databasehandler import DatabaseHandler

logger = logging.getLogger('polar.exploratory_data_analysis.distribution')


def perform(db_handler: DatabaseHandler, configuration):
    """
    Illustrates separate histograms of true wind speed, true wind angle and boat speed.
    :param db_handler: database handler with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    cursor = db_handler.connection.cursor()
    __print_twa(cursor, configuration)
    __print_twa_polar(cursor, configuration)
    __print_twa_one_sail(cursor, configuration)
    __print_tws(cursor, configuration)
    __print_tws_one_sail(cursor, configuration)
    cursor.close()


def __load_twa_histogram_values_count(cursor, twa_area, configuration):
    parameters = (twa_area, twa_area, configuration['from_timestamp'], configuration['to_timestamp'])
    cursor.execute('SELECT count(twa) FROM logger'
                   ' WHERE bsp IS NOT NULL'
                   ' AND twa IS NOT NULL'
                   ' AND tws IS NOT NULL'
                   ' AND abs(twa) BETWEEN ? AND ?+1'
                   ' AND ((timestamp > ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    val = cursor.fetchall()
    return val


def __load_twa_histogram_values(cursor, configuration):
    parameters = (configuration['from_timestamp'], configuration['to_timestamp'])
    cursor.execute('SELECT twa FROM logger'
                   ' WHERE bsp IS NOT NULL'
                   ' AND twa IS NOT NULL'
                   ' AND tws IS NOT NULL'
                   ' AND ((timestamp > ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    val = cursor.fetchall()
    return val


def __load_tws_histogram_values(cursor, tws_area, configuration):
    parameters = (tws_area, tws_area, configuration['from_timestamp'], configuration['to_timestamp'])
    cursor.execute('SELECT count(tws) FROM logger'
                   ' WHERE bsp IS NOT NULL'
                   ' AND tws IS NOT NULL'
                   ' AND tws IS NOT NULL'
                   ' AND abs(tws) >= ?'
                   ' AND abs(tws) < ?+1'
                   ' AND ((timestamp > ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    val = cursor.fetchall()
    return val


def __polar_plot(count_array, title, file_path, show_title, logarithmic=True):
    title_string = 'Rate of occurrences of data points; ' + title
    plt.figure()
    ax = plt.subplot(111, projection='polar')
    width = 2 * numpy.pi / len(count_array)
    theta = numpy.arange(0, 2 * numpy.pi, width)

    if logarithmic:
        for i, count in enumerate(count_array):
            if count is 0:
                count_array[i] = 1
        radii = numpy.log(count_array)
    else:
        radii = count_array

    title_string += ',_logarithmic'
    bars = plt.bar(theta, radii, width=width, bottom=1)

    for r, bar in zip(radii, bars):
        bar.set_facecolor(plt.cm.jet(numpy.log(r) / 10.))
        bar.set_alpha(0.5)

    if show_title:
        ax.set_title(title_string, va='bottom')
    ax.set_theta_direction(-1)
    ax.set_theta_offset(numpy.pi / 2.0)
    ax.set_xticklabels(TWA_TICK_LABELS)  # True Wind Angle

    ax.set_rscale('log', nonposr='clip')
    ax.set_rlim(1, max(radii))
    ax.grid(True)

    plt.draw()
    plt.savefig(file_path)


def __plot(count_array, title, file_path, show_title=True, show_wind_ticks=False):
    plt.figure()
    if show_title:
        plt.title("Histogram_of_data_points;_" + title)
    plt.hist(count_array, bins=180)
    plt.xlabel(title)
    if show_wind_ticks:
        ticks = range(-180, 181, 45)
        plt.xticks(ticks, ticks)
    plt.ylabel("Frequency")
    plt.grid(True)
    plt.draw()
    plt.savefig(file_path)


def __print_twa_one_sail(cursor, configuration):
    histogram_values = fit3D.load_3d_values_with_tuples(cursor, configuration)
    histogram_values_180_unpacked = wind_converter.wind_angle_convert_to_range_180_to_180_list(histogram_values, tuple_index=1)
    logger.debug("count_True_Wind_Angle:_" + str(len(histogram_values_180_unpacked)))
    title = "True_Wind_Angle_[degree]"
    __plot(histogram_values_180_unpacked, title, configuration['output_path'] + "plots/distribution_twa.png", configuration['
        show_title_in_plot'], show_wind_ticks=True)
    plt.show()


def __get_values_of_tuple(tuple_list, tuple_index):
    unpacked_tuple_values = []
    for tuple in tuple_list:
        unpacked_tuple_values.append(tuple[tuple_index])
    return unpacked_tuple_values


def __print_tws_one_sail(cursor, configuration):
    histogram_values = fit3D.load_3d_values_with_tuples(cursor, configuration)
    histogram_values_unpacked = __get_values_of_tuple(histogram_values, tuple_index=0)
    logger.debug("count_True_Wind_Speed:_" + str(len(histogram_values_unpacked)))
    title = "True_Wind_Speed_[kn]"
    __plot(histogram_values_unpacked, title, configuration['output_path'] + "plots/distribution_tws.png", configuration['
        show_title_in_plot'], show_wind_ticks=False)
    plt.show()


def __print_twa(cursor, configuration):
    histogram_values = __load_twa_histogram_values(cursor, configuration)
    histogram_values_180_unpacked = wind_converter.wind_angle_convert_to_range_180_to_180_list(histogram_values)
    logger.debug("count_True_Wind_Angle:_" + str(len(histogram_values_180_unpacked)))
    title = "True_Wind_Angle_[degree]"
    __plot(histogram_values_180_unpacked, title, configuration['output_path'] + "plots/distribution_twa.png", configuration['
        show_title_in_plot'], show_wind_ticks=True)
    plt.show()


def __print_twa_polar(cursor, configuration):
    sum = 0
    values_twa = []
    for twa in range(configuration['from_wind_angle'], configuration['to_wind_angle'], 1):
        count_result = __load_twa_histogram_values_count(cursor, twa, configuration)
        count = count_result[0][0]
        sum += count
        values_twa.append(count)
    logger.debug("count True Wind Angle: " + str(sum))
    title = "True Wind Angle [degree]"
    __polar_plot(values_twa, title, configuration['output_path'] + "plots/distribution_twa_polar.png", False, configuration['
        show_title_in_plot'])
    plt.show()


def __print_tws(cursor, configuration):
    sum = 0
    values_tws = []
    for tws in range(configuration['from_wind_speed'], configuration['to_wind_speed'], 1):
        count_result = __load_tws_histogram_values(cursor, tws, configuration)
        count = count_result[0][0]
        sum += count
        values_tws.append(count)
    logger.debug("count True Wind Speed: " + str(sum))
    __plot(values_tws, "True Wind Speed [kn]", configuration['output_path'] + "plots/distribution_tws.png", configuration['
        show_title_in_plot'], show_wind_ticks=False)
    plt.show()
```

Listing A.13: exploratory˙data˙analysis/occurrence˙plot.py

```
"""
Illustrates the frequency of data points in a polar diagram.

Performs exploratory data analysis of the data.
Loads data from the database as requested and generates a polar plot.
It shows a heat map of the frequency of data points at narrow areas of wind angle and wind speed.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import numpy
from matplotlib import pyplot as plt
from matplotlib.collections import PolyCollection

from polar_diagram_generator.converter import wind_converter
from polar_diagram_generator.util.strings import TWA_TICK_LABELS
from polar_diagram_generator.workspace.databasehandler import DatabaseHandler

SPEED_DIFF = 1

ANGLE_DIFF = 3.5

SHOW_WHOLE_LINES = False
MAX_ERROR = 10
PREDICTION_PROBABILITY = 0.10


def perform(db_handler: DatabaseHandler, configuration):
    """
    Illustrates the frequency of data points in a polar diagram.
    :param db_handler: database handler with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    cursor = db_handler.connection.cursor()

    fig = plt.figure()
    ax = plt.subplot(111, projection='polar')
    ax.set_title('Frequency of data', va='bottom')
    ax.set_theta_direction(-1)
    ax.set_theta_offset(numpy.pi / 2.0)
    ax.set_xticklabels(TWA_TICK_LABELS)  # Twa
    ax.set_yticks([2., 4., 6., 8., 10., 12., 14., 16., 18., 20.])
    totalcount = __count_values_limited_to_sail(25 - ANGLE_DIFF, 180 + ANGLE_DIFF, 6 - SPEED_DIFF, 20 + SPEED_DIFF, cursor,
         configuration)

    for index, speed in enumerate([6., 8., 10., 12., 14., 16., 18., 20.]):
        for angle in range(24, 337, 8):
            count = __count_values_limited_to_sail(angle - ANGLE_DIFF, angle + ANGLE_DIFF, speed - SPEED_DIFF, speed +
                 SPEED_DIFF, cursor, configuration)

            __plot_poly(ax, angle-3.4, angle+3.4, speed-0.9, speed+0.9, __get_color(count, totalcount))

    ax.axis([0, 2*numpy.pi, 0, 22])

    plt.draw()
    plt.savefig(configuration['output_path'] + 'plots/occurrences_polar.png')
    plt.show()

    cursor.close()


def __count_values(from_twa, to_twa, from_tws, to_tws, cursor, configuration):
    parameters = (from_twa, to_twa, from_tws, to_tws,
                  configuration['from_timestamp'],
                  configuration['to_timestamp'],
                  configuration['from_mark'],
                  configuration['to_mark'])
    cursor.execute('SELECT␣count(*)␣FROM␣logger'
                   '␣WHERE␣bsp␣IS␣NOT␣NULL'
                   '␣AND␣twa␣IS␣NOT␣NULL'
                   '␣AND␣tws␣IS␣NOT␣NULL'
                   '␣AND␣twa␣>=␣?'
                   '␣AND␣twa␣<=␣?'
                   '␣AND␣tws␣>=␣?'
                   '␣AND␣tws␣<=␣?'
                   '␣AND␣outlier␣IS␣NOT␣1'
                   '␣AND␣((timestamp␣>␣?␣AND␣timestamp␣<=␣?)␣OR␣timestamp␣IS␣NULL)'
                   '␣AND␣((mark␣>␣?␣AND␣mark␣<=␣?)␣OR␣mark␣IS␣NULL)'
                   # ' ORDER BY RANDOM()'
                   # ' LIMIT 15000'
                   , parameters)
    features = cursor.fetchall()
    return features[0][0]


def __count_values_limited_to_sail(fromTwa, toTwa, fromTws, toTws, cursor, configuration):
    if configuration['mark_tuples'] is None or len(configuration['mark_tuples']) <= 0:
        raise RuntimeError("mark_tuples␣not␣initialized␣or␣empty")

    parameters = (fromTwa, toTwa, fromTws, toTws,
                  configuration['from_boat_speed'])
    query = 'SELECT␣count(*)␣FROM␣logger' \
            '␣WHERE␣bsp␣IS␣NOT␣NULL' \
            '␣AND␣twa␣IS␣NOT␣NULL' \
            '␣AND␣tws␣IS␣NOT␣NULL' \
            '␣AND␣twa␣>=␣?' \
            '␣AND␣twa␣<␣?' \
            '␣AND␣tws␣>=␣?' \
            '␣AND␣tws␣<␣?' \
            '␣AND␣bsp␣>=␣?' \
            '␣AND␣outlier␣IS␣NOT␣1' \
            '␣AND␣('
    first = True
    for start, end in configuration['mark_tuples']:
        if not first:
            query += '␣OR␣'
        first = False
        query += '␣mark␣BETWEEN␣' + str(start) + '␣AND␣' + str(end) + '␣'

    query += '␣)'
    cursor.execute(query, parameters)
    features = cursor.fetchall()
    return features[0][0]


def __plot_poly(axes, from_twa, to_twa, from_tws, to_tws, linecolor):

    from_twa_polar = wind_converter.wind_angle_cart_2_pol(from_twa)
    to_twa_polar = wind_converter.wind_angle_cart_2_pol(to_twa)

    verts_pred = [(from_twa_polar, from_tws), (from_twa_polar, to_tws), (to_twa_polar, to_tws), (to_twa_polar, from_tws)]
    poly = PolyCollection([verts_pred], closed=True,
                          label="PI_(%g)" % from_tws)
    poly.set_facecolor(linecolor)
    axes.add_collection(poly)


def __get_color(count, totalcount):
    if count > 600:
        return [0.0, 1.0, 0.0, 1.0]
    if count > 500:
        return [0.0, 1.0, 0.0, 0.8]
    if count > 400:
        return [0.0, 1.0, 0.0, 0.6]
    if count > 300:
        return [0.0, 1.0, 0.0, 0.4]
    if count > 150:
        return [0.0, 0.6, 0.0, 0.4]
    if count > 100:
        return [0.0, 0.3, 0.0, 0.4]
    if count > 80:
        return [0.3, 0.0, 0.0, 0.4]
    if count > 40:
        return [0.6, 0.0, 0.0, 0.4]
    if count > 20:
        return [1.0, 0.0, 0.0, 0.4]
    if count > 10:
        return [1.0, 0.0, 0.0, 0.6]
    return [1.0, 0.0, 0.0, 0.8]
```

Listing A.14: exploratory_data_analysis/speed_trend.py

```
"""
Illustrates the trend of true wind speed and boat speed.

Performs exploratory data analysis of the data.
Loads data from the database as requested and generates a plot.
It shows the process of both speed values to analyze abnormalities.
Can also highlight the areas of the selected sail.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import matplotlib.pylab as plt


def perform(cursor, configuration):
    """
    Illustrates the trend of true wind speed and boat speed.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    __print_tws(cursor, configuration)


def plot(mark, bsp, tws, title, file_path=None, x_label="Time_[s]", y_label="Speed_[knots]",
         legend=["Boat_Speed", "True_Wind_Speed"]):
    """
    Plots a scatter plot with the requested values.
    :param mark: list of points on the timeline
    :param bsp: values of the BSP. Same length as mark
    :param tws: values of the TWS. Same length as mark
    :param title: title of the plot
    :param file_path: path where the plot should be saved
    :param x_label: name of the time axis
    :param y_label: name of the y axis
    :param legend: array with size 2; name of the features
    """
    plt.figure()
    plt.title(title)
    plt.scatter(mark, bsp, marker='.')
    plt.scatter(mark, tws, marker='.')
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.legend(legend)
    plt.grid(True)
    plt.draw()
    if file_path is not None:
        plt.savefig(file_path)


def __load_speed_values(cursor, configuration):
    parameters = ()
    cursor.execute('SELECT mark FROM logger'
                   , parameters)
    mark = cursor.fetchall()
    cursor.execute('SELECT bsp FROM logger'
                   , parameters)
    bsp = cursor.fetchall()
    cursor.execute('SELECT tws FROM logger'
                   , parameters)
    tws = cursor.fetchall()
    return mark, bsp, tws


def load_speed_values_of_range(cursor, configuration):
    parameters = (configuration['from_mark'], configuration['to_mark'], configuration['from_timestamp'],
                  configuration['to_timestamp'])
    cursor.execute('SELECT bsp FROM logger'
                   ' WHERE ((mark >= ? AND mark <= ?) OR mark IS NULL)'
                   ' AND ((timestamp >= ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    bsp = cursor.fetchall()
    cursor.execute('SELECT tws FROM logger'
                   ' WHERE ((mark >= ? AND mark <= ?) OR mark IS NULL)'
                   ' AND ((timestamp >= ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    tws = cursor.fetchall()
    cursor.execute('SELECT mark FROM logger'
                   ' WHERE ((mark >= ? AND mark <= ?) OR mark IS NULL)'
                   ' AND ((timestamp >= ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    mark = cursor.fetchall()
    return mark, bsp, tws


def __print_tws(cursor, configuration):
    mark, bsp, tws = __load_speed_values(cursor, configuration)
    if configuration['show_title_in_plot']:
        plot_title = "Speed trend"
    else:
        plot_title = ""
    plot(mark, bsp, tws, plot_title, configuration['output_path'] + "plots/speed_trend.png")
    __add_active_logbook_indicator(configuration)
    plt.show()


def __add_active_logbook_indicator(configuration):
    from_mark = configuration['from_mark']
    to_mark_ = configuration['to_mark']
    if from_mark == 0 and to_mark_ == 99999999999:
        return
    active_width = to_mark_ - from_mark
    currentAxis = plt.gca()
    currentAxis.add_patch(
        plt.Rectangle((from_mark, 0), active_width, max(configuration['to_wind_speed'], configuration['to_boat_speed']),
                      facecolor="grey", fill="grey", alpha=0.4))
```

Listing A.15: exploratory˙data˙analysis/wind˙angle˙trend.py

```
"""
Illustrates the trend of true wind angle.

Performs exploratory data analysis of the data.
Loads data from the database as requested and generates a plot.
It shows the process of the true wind angle to analyze abnormalities.
Can also highlight the areas of the selected sail.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import matplotlib.pylab as plt
```

```
def perform(cursor, configuration):
    """
    Illustrates the trend of true wind angle.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    __print_twa(cursor, configuration)


def wind_angle_values(cursor, configuration):
    """
    Loads data from the database without limitations.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: Not used
    :return: a tuple with two arrays mark and TWA
    """
    parameters = ()
    cursor.execute('SELECT mark FROM logger'
                   , parameters)
    mark = cursor.fetchall()
    cursor.execute('SELECT twa FROM logger'
                   , parameters)
    twa = cursor.fetchall()
    return mark, twa


def wind_angle_values_of_range(cursor, configuration):
    """
    Loads data from the database with limitations.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    :return: a tuple with two arrays mark and TWA
    """
    parameters = (configuration['from_mark'], configuration['to_mark'], configuration['from_timestamp'],
                  configuration['to_timestamp'])
    cursor.execute('SELECT mark FROM logger'
                   ' WHERE ((mark > ? AND mark <= ?) OR mark IS NULL)'
                   ' AND ((timestamp > ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    mark = cursor.fetchall()
    cursor.execute('SELECT twa FROM logger'
                   ' WHERE ((mark > ? AND mark <= ?) OR mark IS NULL)'
                   ' AND ((timestamp > ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    twa = cursor.fetchall()
    return mark, twa


def plot(mark, twa, title, file_path):
    plt.figure(figsize=(15, 6))
    if title is not None:
        plt.title(title)
    plt.scatter(mark, twa, marker='.')
    plt.xlabel("Time [seconds]")
    plt.ylabel("True Wind Angle [degree]")
    plt.legend(["True Wind Angle"])
    ticks = range(-180, 181, 45)
    plt.yticks(ticks, ticks)
    plt.grid(True)
    plt.draw()
    plt.savefig(file_path)


def __print_twa(cursor, configuration):
    mark, twa = wind_angle_values(cursor, configuration)
    if configuration['show_title_in_plot']:
        title = "Scatter Diagram of True Wind Angle"
    else:
        title = None
    plot(mark, twa, title, configuration['output_path'] + "plots/wind_angle_trend.png")
    add_active_logbook_indicator(configuration['from_mark'], configuration['to_mark'])
    plt.show()


def add_active_logbook_indicator(from_mark, to_mark_):
    """
    Highlights an area in the plot that indicates that this area is included in the data analysis.
    :param from_mark: begin of the selected range
    :param to_mark_: end of the selected range
    :return: None
    """
    if from_mark == 0 and to_mark_ == 99999999999:
        return
    active_width = to_mark_ - from_mark
    current_axis = plt.gca()
    current_axis.add_patch(plt.Rectangle((from_mark, -180), active_width, 360, facecolor="grey", fill="grey", alpha=0.4))
```

Listing A.16: data˙analysis/fit2D˙angle.py

```
"""
Performs data analysis of true wind angle and boat speed.

Uses the extendable interface of scikit-learn to fit 2-dimensional models to the data.
Loads data from the database and performs pre-processing on the data.
Supports various types of estimators for the optimization problem.
Generates plots from the result.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging
import timeit
from random import randrange

import numpy as np
from matplotlib import pyplot as plt
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MaxAbsScaler
from statsmodels.stats.stattools import durbin_watson

from polar_diagram_generator.converter import wind_converter
from polar_diagram_generator.data_analysis import fit3D
from polar_diagram_generator.data_analysis.estimator.odr_estimator_2d import ODREstimator2d
from polar_diagram_generator.data_analysis.models import models2d

logger = logging.getLogger('polar.data_analysis.fit')


def perform(cursor, configuration, true_wind_speed, true_wind_speed_range = 1):
    """
    Performs data analysis of true wind angle and boat speed.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    :param true_wind_speed: TWS around which the data should be trained
    :param true_wind_speed_range: optional, TWS deviation around which the data should be trained
    :return: None
    """
    tws_from = true_wind_speed-true_wind_speed_range
    tws_to = true_wind_speed+true_wind_speed_range

    X_list, y_list = __load_2d_values(cursor, tws_from, tws_to, configuration)
    __fit_and_plot(X_list, y_list, true_wind_speed, configuration, x_axis_label="True Wind Angle [degree]")
    plt.show()


def __load_2d_values(cursor, tws_from, tws_to, configuration):
    previous_from_wind_speed = configuration["from_wind_speed"]
    previous_to_wind_speed = configuration["to_wind_speed"]
    configuration["from_wind_speed"] = tws_from
    configuration["to_wind_speed"] = tws_to

    tuples = fit3D.load_3d_values_with_tuples(cursor, configuration)
    configuration["from_wind_speed"] = previous_from_wind_speed
    configuration["to_wind_speed"] = previous_to_wind_speed
    bsp_values = __get_values_of_tuple(tuples, tuple_index=3)
    twa_values = wind_converter.wind_angle_convert_to_range_180_to_180_list(tuples, tuple_index=2)
    return twa_values, bsp_values


def __fit_and_plot(X_list, y_list, speed, configuration, x_axis_label):
    logger.debug("\n\nFitting data for %d knots..." % speed)
    X = np.asarray(X_list)
    y = np.asarray(y_list)

    random_state = randrange(1000)
```

```
logger.debug("random_state: %d" % random_state)
X_train, X_test, y_train, y_test = train_test_split(X.reshape(-1, 1), y.reshape(-1, 1), test_size=0.3, random_state=
      random_state)

if len(X_train) == 0:
    logger.warning("No values for this range!")
    return

logger.debug("# training set: %d\n    # test set: %d" % (len(X_train), len(X_test)))

X_scaler = MaxAbsScaler()
X_scaler.fit(X_train)
X_train = X_scaler.transform(X_train)
X_test = X_scaler.transform(X_test)

y_scaler = MaxAbsScaler()
y_scaler.fit(y_train)
y_train = y_scaler.transform(y_train)
y_test = y_scaler.transform(y_test)

estimators = [
    ('Nonlinear ODR - 2 degree Polynomial', ODREstimator2d(models2d.polynomial_2_degrees, [1.0, 1.0, 1.0], 1.0, 1.0),
         models2d.polynomial_2_degrees),
    ('Nonlinear ODR - 3 degree Polynomial',
     ODREstimator2d(models2d.polynomial_3_degrees, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.polynomial_3_degrees),
    ('Nonlinear ODR - 4 degree Polynomial',
     ODREstimator2d(models2d.polynomial_4_degrees, [1.0, 1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.polynomial_4_degrees),
    ('Nonlinear ODR - 6 degree Polynomial',
     ODREstimator2d(models2d.polynomial_6_degrees, [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
          polynomial_6_degrees),
    ('Nonlinear ODR - Inverted Parabola', ODREstimator2d(models2d.inverted_parabola, [1.0, 1.0, 1.0], 1.0, 1.0), models2d.
         inverted_parabola),
    ('Nonlinear ODR - Concave with Downturn', ODREstimator2d(models2d.concave_with_downturn, [1.0, 1.0, 1.0], 1.0, 1.0),
         models2d.concave_with_downturn),
    ('Nonlinear ODR - Concave with Saturation Limit',
     ODREstimator2d(models2d.concave_with_saturation_limit, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
          concave_with_saturation_limit),
    ('Nonlinear ODR - Concave with Saturation Limit and Downturn',
     ODREstimator2d(models2d.concave_with_saturation_limit_and_downturn, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
          concave_with_saturation_limit_and_downturn),
    ('Nonlinear ODR - S-shaped with Saturation Limit',
     ODREstimator2d(models2d.s_shaped_with_saturation_limit, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
          s_shaped_with_saturation_limit),
    ('Nonlinear ODR - S-shaped with Saturation Limit and Downturn',
     ODREstimator2d(models2d.s_shaped_with_saturation_limit_and_downturn, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
          s_shaped_with_saturation_limit_and_downturn),
    ('Nonlinear ODR - Gompertz Model', ODREstimator2d(models2d.gompertz_model, [1.0, 1.0, 1.0], 1.0, 1.0), models2d.
         gompertz_model),
    ('Nonlinear ODR - Gaussian Model', ODREstimator2d(models2d.gaussian_model, [5.0, 1.0, 1.0], 1.0, 1.0), models2d.
         gaussian_model),
    ('Nonlinear ODR - Gaussian Model with Offset',
     ODREstimator2d(models2d.gaussian_model_with_offset, [5.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
          gaussian_model_with_offset),
    ('Nonlinear ODR - Gaussian Mixture Model',
     ODREstimator2d(models2d.gmm_model, [5.0, 1.0, 1.0, 5.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.gmm_model),
    ]
x_plot = np.linspace(X_train.min(), X_train.max())

result_output_list = []
for title, this_X, this_y, this_X_test, this_y_test in [
    ('Compare Boat Speed and True Wind Angle at ' + str(speed) + ' degree True Wind Speed', X_train, y_train, X_test,
         y_test)
]:
    fig, ticks = __prepare_plot(X_scaler, configuration, estimators, this_X_test, this_y_test, title, x_axis_label,
                                y_scaler)

    for name, estimator, model in estimators:
        logger.debug('\n=================================================\n')
        start = timeit.default_timer()
        estimator.fit(this_X, this_y)
        stop = timeit.default_timer()
        runtime_seconds = stop - start
        logger.debug('\n%s: time = %.5f' % (name, runtime_seconds))
        estimator_train_prediction = estimator.predict(this_X)
        mean_squared_error_train_scaled = metrics.mean_squared_error(y_train, estimator_train_prediction)
        mean_squared_error_train = y_scaler.inverse_transform(np.array([mean_squared_error_train_scaled]).reshape(-1, 1))
        logger.debug('%s: mean_squared_error_train = %.3f' % (name, mean_squared_error_train))
        estimator_test_prediction = estimator.predict(this_X_test)
        mean_squared_error_test_scaled = metrics.mean_squared_error(y_test, estimator_test_prediction)
        mean_squared_error_test = y_scaler.inverse_transform(np.array([mean_squared_error_test_scaled]).reshape(-1, 1))
```

```
            logger.debug('%s: mean_squared_error_test = %.3f' % (name, mean_squared_error_test))

            chi_squared_train_value = y_scaler.inverse_transform([estimator.output.res_var * len(X_train)])[0]

            r2 = metrics.r2_score(y_test, estimator_test_prediction)
            logger.debug('%s: r2_score = %.3f' % (name, r2))
            mean_absolute_error = metrics.mean_absolute_error(y_test, estimator_test_prediction)
            mean = np.mean(y_test)
            logger.debug('%s: mean_absolute_error = %.3f, mean = %.3f, %.3f%%' % (name, y_scaler.inverse_transform(np.array([
                 mean_absolute_error]).reshape(-1, 1)), y_scaler.inverse_transform(np.array([mean]).reshape(-1, 1)), 100/mean*
                 mean_absolute_error))
            median_absolute_error_test = metrics.median_absolute_error(y_test, estimator_test_prediction)
            median = np.median(y_test)
            logger.debug('%s: median_absolute_error_test = %.3f, median = %.3f, %.3f%%' % (name, y_scaler.inverse_transform(np.
                 array([median_absolute_error_test]).reshape(-1, 1)), y_scaler.inverse_transform(np.array([median]).reshape(-1,
                  1)), 100/median*median_absolute_error_test))

            y_plot = estimator.predict(x_plot[:, np.newaxis])
            label = '%s' % (name)
            __add_to_plot(X_scaler, label, x_plot, y_plot, y_scaler)

            differences = y_test - estimator.predict(X_test).reshape(-1, 1) #reshape to get [nx1] instead of [nxn]
            durbin_watson_value = durbin_watson(differences)
            logger.debug("Durbin/Watson: " + str(durbin_watson_value))

            uniform_weight_test = np.ones(len(y_test))

            logger.debug("\n\n\n\n\n----------------- TEST DATA UNIFORM -------------------")
            odr_estimator_test_data = ODREstimator2d(model, estimator.popt, uniform_weight_test, uniform_weight_test,
                 max_iterations=0)
            odr_estimator_test_data.fit(X_test, y_test)

            chi_squared_test_value = y_scaler.inverse_transform([odr_estimator_test_data.output.res_var * len(X_test)])[0]
            sum_square_test_uniform = odr_estimator_test_data.output.sum_square
            logger.debug("--------------------------------------------------------------")


            if len(estimators) == 1:
                __add_labels_and_residual_plot(X_scaler, X_test, configuration, differences, fig, name, ticks, title,
                                               x_axis_label, y_scaler)

            result_output_list.append((name, runtime_seconds, mean_squared_error_train, mean_squared_error_test,
                 durbin_watson_value, chi_squared_train_value, chi_squared_test_value))

        __configure_legend()

    __save_plot(configuration, speed)

    logger.debug("\n\n\n============== Summary ===============\nNumber of estimators in this run: %d" % len(estimators))
    for name, estimator, model in estimators:
        logger.debug("- %s" % name)

    logger.debug("\n---- LaTeX Result Table ----\n")
    logger.debug("%s & %s & %s & %s & %s" % ("Name", "Runtime [s]", "Durbin/Watson", "\chi^2_{train}", "\chi^2_{test}"))
    for (name, runtime_seconds, mse_train, mse_test, durbin_watson_value, chi_squared_train_value, chi_squared_test_value) in
          result_output_list:
        logger.debug("%s & $%.2f~s$ & $%.4f$ & $%.2f$ & $%.2f$ \\\\" % (name.replace("Nonlinear ODR - ", ""), runtime_seconds,
             durbin_watson_value, chi_squared_train_value, chi_squared_test_value))


def __get_values_of_tuple(tuple_list, tuple_index):
    unpacked_tuple_values = []
    for tuple in tuple_list:
        unpacked_tuple_values.append(tuple[tuple_index])
    return unpacked_tuple_values


def __save_plot(configuration, speed):
    plt.savefig(configuration['output_path'] + "plots/fit_2d_angle_" + str(speed) + ".png")


def __configure_legend():
    plt.legend(loc='best', frameon=False)


def __add_labels_and_residual_plot(X_scaler, X_test, configuration, differences, fig, name, ticks, title, x_axis_label,
                                   y_scaler):
    if configuration['show_title_in_plot']:
        plt.title(name + " - " + title)
    frame_residual = fig.add_axes((.1, .07, .8, .2))
    plt.xticks(ticks, ticks)
    plt.xlim(0, 180)
    plt.xlabel(x_axis_label)
    plt.ylabel("Boat␣Speed␣Residuals␣[kn]")
    frame_residual.plot(X_scaler.inverse_transform(X_test), y_scaler.inverse_transform(differences), 'rx')
    plt.grid()


def __add_to_plot(X_scaler, label, x_plot, y_plot, y_scaler):
    plt.plot(X_scaler.inverse_transform(x_plot), y_scaler.inverse_transform(y_plot), label=label)


def __prepare_plot(X_scaler, configuration, estimators, this_X_test, this_y_test, title, x_axis_label, y_scaler):
    fig = plt.figure(figsize=(10, 8))
    if len(estimators) == 1:
        fig.add_axes((.1, .35, .8, .55))
    plt.xlabel(x_axis_label)
    ticks = range(-180, 181, 45)
    plt.xticks(ticks, ticks)
    plt.ylim(0, configuration['to_boat_speed'])
    plt.ylabel("Boat␣Speed␣[kn]")
    if configuration['show_title_in_plot']:
        plt.title(title)
    plt.plot(X_scaler.inverse_transform(this_X_test), y_scaler.inverse_transform(this_y_test), 'k+')  # test set
    plt.xlim(0, 180)
    plt.grid()
    return fig, ticks
```

Listing A.17: data˙analysis/fit2D˙speed.py

```
"""
Performs data analysis of true wind speed and boat speed.

Uses the extendable interface of scikit-learn to fit 2-dimensional models to the data.
Loads data from the database and performs pre-processing on the data.
Supports various types of estimators for the optimization problem.
Generates plots from the result.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging
import timeit

import numpy as np
from matplotlib import pyplot as plt
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MaxAbsScaler
from statsmodels.stats.stattools import durbin_watson

from polar_diagram_generator.data_analysis.estimator.odr_estimator_2d import ODREstimator2d
from polar_diagram_generator.data_analysis.models import models2d

logger = logging.getLogger('polar.data_analysis.fit')


def __load_2d_values(cursor, twa, true_wind_angle_range, configuration):
    parameters = (float(twa)-true_wind_angle_range,
                  float(twa)+true_wind_angle_range,
                  configuration['from_boat_speed'],
                  configuration['to_boat_speed'],
                  configuration['from_wind_speed'],
                  configuration['to_wind_speed'],
                  configuration['from_timestamp'],
                  configuration['to_timestamp'],
                  configuration['from_mark'],
                  configuration['to_mark'])
    cursor.execute('SELECT␣bsp␣FROM␣logger'
                   '␣WHERE␣twa␣>=␣?'
                   '␣AND␣twa␣<␣?'
                   '␣AND␣outlier␣IS␣NOT␣42'
                   '␣AND␣twa␣IS␣NOT␣NULL'
                   '␣AND␣bsp␣IS␣NOT␣NULL␣AND␣(bsp␣>␣?␣AND␣bsp␣<=␣?)'
                   '␣AND␣tws␣IS␣NOT␣NULL␣AND␣(tws␣>␣?␣AND␣tws␣<=␣?)'
                   '␣AND␣((timestamp␣>␣?␣AND␣timestamp␣<=␣?)␣OR␣timestamp␣IS␣NULL)'
                   '␣AND␣((mark␣>␣?␣AND␣mark␣<=␣?)␣OR␣mark␣IS␣NULL)'
                   # ' AND (mark % 10 IS 0)'
                   , parameters)
    bsp = cursor.fetchall()
    cursor.execute('SELECT tws FROM logger'
                   ' WHERE twa >= ?'
                   ' AND twa < ?'
                   ' AND outlier IS NOT 42'
                   ' AND twa IS NOT NULL'
                   ' AND bsp IS NOT NULL AND (bsp > ? AND bsp <= ?)'
                   ' AND tws IS NOT NULL AND (tws > ? AND tws <= ?)'
                   ' AND ((timestamp > ? AND timestamp <= ?) OR timestamp IS NULL)'
                   ' AND ((mark > ? AND mark <= ?) OR mark IS NULL)'
                   # ' AND (mark % 10 IS 0)'
                   , parameters)
    tws_values = cursor.fetchall()
    return tws_values, bsp


def perform(cursor, configuration, true_wind_angle, true_wind_angle_range=0.5):
    """
    Performs data analysis of true wind speed and boat speed.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    :param true_wind_angle: TWA around which the data should be trained
    :param true_wind_angle_range: optional, TWA deviation around which the data should be trained
    """
    for angle in range(true_wind_angle, true_wind_angle+1, 5):
        X_list, y_list = __load_2d_values(cursor, angle, true_wind_angle_range, configuration)
        __fit_and_plot(X_list, y_list, angle, configuration, x_axis_label="True Wind Speed [kn]")
    plt.show()


def __fit_and_plot(X_list, y_list, angle, configuration, x_axis_label):
    logger.debug("\n\nFitting data around %d degree ..." % angle)
    X = np.asarray(X_list)
    y = np.asarray(y_list)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=456)

    if len(X_train) == 0:
        logger.warning("No values for this range!")
        return

    logger.debug("# training set: %d\n    # test set: %d" % (len(X_train), len(X_test)))

    X_scaler = MaxAbsScaler()
    X_scaler.fit(X_train)
    X_train = X_scaler.transform(X_train)
    X_test = X_scaler.transform(X_test)

    y_scaler = MaxAbsScaler()
    y_scaler.fit(y_train)
    y_train = y_scaler.transform(y_train)
    y_test = y_scaler.transform(y_test)

    estimators = [
        ('Nonlinear ODR - 2 degree Polynomial', ODREstimator2d(models2d.polynomial_2_degrees, [1.0, 1.0, 1.0], 1.0, 1.0),
             models2d.polynomial_2_degrees),
        ('Nonlinear ODR - 3 degree Polynomial',
         ODREstimator2d(models2d.polynomial_3_degrees, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.polynomial_3_degrees),
        ('Nonlinear ODR - 4 degree Polynomial',
         ODREstimator2d(models2d.polynomial_4_degrees, [1.0, 1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.polynomial_4_degrees),
        ('Nonlinear ODR - 6 degree Polynomial',
         ODREstimator2d(models2d.polynomial_6_degrees, [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
              polynomial_6_degrees),
        ('Nonlinear ODR - Inverted Parabola', ODREstimator2d(models2d.inverted_parabola, [1.0, 1.0, 1.0], 1.0, 1.0), models2d.
             inverted_parabola),
        ('Nonlinear ODR - Concave with Downturn', ODREstimator2d(models2d.concave_with_downturn, [1.0, 1.0, 1.0], 1.0, 1.0),
             models2d.concave_with_downturn),
        ('Nonlinear ODR - Concave with Saturation Limit',
         ODREstimator2d(models2d.concave_with_saturation_limit, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
              concave_with_saturation_limit),
        ('Nonlinear ODR - Concave with Saturation Limit and Downturn',
         ODREstimator2d(models2d.concave_with_saturation_limit_and_downturn, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
              concave_with_saturation_limit_and_downturn),
        ('Nonlinear ODR - S-shaped with Saturation Limit',
         ODREstimator2d(models2d.s_shaped_with_saturation_limit, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
              s_shaped_with_saturation_limit),
        ('Nonlinear ODR - S-shaped with Saturation Limit and Downturn',
         ODREstimator2d(models2d.s_shaped_with_saturation_limit_and_downturn, [1.0, 1.0, 1.0, 1.0], 1.0, 1.0), models2d.
```

```
                s_shaped_with_saturation_limit_and_downturn),
        ('Nonlinear ODR - Gompertz Model', ODREstimator2d(models2d.gompertz_model, [1.0, 1.0, 1.0], 1.0, 1.0), models2d.
             gompertz_model),
        ]
    x_plot = np.linspace(X_train.min(), X_train.max())

    result_output_list = []
    for title, this_X, this_y, this_X_test, this_y_test in [
        ('Compare Boat Speed and True Wind Speed at ' + str(angle) + ' degree True Wind Angle', X_train, y_train, X_test,
             y_test)
    ]:
        fig = __prepare_plot(X_scaler, configuration, estimators, this_X_test, this_y_test, title, x_axis_label,
                             y_scaler)

        for name, estimator, model in estimators:
            logger.debug('\n====================================================\n')
            start = timeit.default_timer()
            estimator.fit(this_X, this_y)
            stop = timeit.default_timer()
            runtime_seconds = stop - start
            logger.debug('\n%s: time = %.5f' % (name, runtime_seconds))
            estimator_train_prediction = estimator.predict(this_X)
            mean_squared_error_train_scaled = metrics.mean_squared_error(y_train, estimator_train_prediction)
            mean_squared_error_train = y_scaler.inverse_transform(np.array([mean_squared_error_train_scaled]).reshape(-1, 1))
            logger.debug('%s: mean_squared_error_train = %.3f' % (name, mean_squared_error_train))
            estimator_test_prediction = estimator.predict(this_X_test)
            mean_squared_error_test_scaled = metrics.mean_squared_error(y_test, estimator_test_prediction)
            mean_squared_error_test = y_scaler.inverse_transform(np.array([mean_squared_error_test_scaled]).reshape(-1, 1))
            logger.debug('%s: mean_squared_error_test = %.3f' % (name, mean_squared_error_test))

            chi_squared_train_value = y_scaler.inverse_transform([estimator.output.res_var * len(X_train)])[0]

            r2 = metrics.r2_score(y_test, estimator_test_prediction)
            logger.debug('%s: r2_score = %.3f' % (name, r2))
            mean_absolute_error = metrics.mean_absolute_error(y_test, estimator_test_prediction)
            mean = np.mean(y_test)
            logger.debug('%s: mean_absolute_error = %.3f, mean = %.3f, %.3f%%' % (name, y_scaler.inverse_transform(np.array([
                  mean_absolute_error]).reshape(-1, 1)), y_scaler.inverse_transform(np.array([mean]).reshape(-1, 1)), 100/mean*
                  mean_absolute_error))
            median_absolute_error_test = metrics.median_absolute_error(y_test, estimator_test_prediction)
            median = np.median(y_test)
            logger.debug('%s: median_absolute_error_test = %.3f, median = %.3f, %.3f%%' % (name, y_scaler.inverse_transform(np.
                  array([median_absolute_error_test]).reshape(-1, 1)), y_scaler.inverse_transform(np.array([median]).reshape(-1,
                   1)), 100/median*median_absolute_error_test))

            y_plot = estimator.predict(x_plot[:, np.newaxis])
            label = '%s' % (name)
            __add_to_plot(X_scaler, label, x_plot, y_plot, y_scaler)

            differences = y_test - estimator.predict(X_test).reshape(-1, 1)
            durbin_watson_value = durbin_watson(differences)
            logger.debug("Durbin/Watson: " + str(durbin_watson_value))

            uniform_weight_test = np.ones(len(y_test))

            logger.debug("\n\n\n\n\n------------------- TEST DATA UNIFORM ---------------------")
            odr_estimator_test_data = ODREstimator2d(model, estimator.popt, uniform_weight_test, uniform_weight_test,
                  max_iterations=0)
            odr_estimator_test_data.fit(X_test, y_test)

            chi_squared_test_value = y_scaler.inverse_transform([odr_estimator_test_data.output.res_var * len(X_test)])[0]
            sum_square_test_uniform = odr_estimator_test_data.output.sum_square
            logger.debug("----------------------------------------------------------")


            if len(estimators) == 1:
                __add_labels_and_residual_plot(X_scaler, X_test, configuration, differences, fig, name, title,
                                               x_axis_label, y_scaler)

            result_output_list.append((name, runtime_seconds, mean_squared_error_train, mean_squared_error_test,
                  durbin_watson_value, chi_squared_train_value, chi_squared_test_value))

        __configure_legend()

    __save_plot(angle, configuration)

    logger.debug("\n\n\n============== Summary ===============\nNumber of estimators in this run: %d" % len(estimators))
    for name, estimator, model in estimators:
        logger.debug("- %s" % name)
    logger.debug("\n---- LaTeX Result Table ----\n")
    logger.debug("%s & %s & %s & %s & %s" % ("Name", "Runtime [s]", "Durbin/Watson", "\chi^2_{train}", "\chi^2_{test}"))
    for (name, runtime_seconds, mse_train, mse_test, durbin_watson_value, chi_squared_train_value, chi_squared_test_value) in
            result_output_list:
        logger.debug("%s & $%.2f~s$ & $%.4f$ & $%.2f$ & $%.2f$ \\\\" % (name.replace("Nonlinear ODR - ", ""), runtime_seconds,
             durbin_watson_value, chi_squared_train_value, chi_squared_test_value))


def __save_plot(angle, configuration):
    plt.savefig(configuration['output_path'] + "plots/fit_2d_" + str(angle) + ".png")


def __configure_legend():
    plt.legend(loc='best', frameon=False)


def __add_labels_and_residual_plot(X_scaler, X_test, configuration, differences, fig, name, title, x_axis_label,
                                   y_scaler):
    if configuration['show_title_in_plot']:
        plt.title(name + " - " + title)
    frame_residual = fig.add_axes((.1, .07, .8, .2))
    plt.xlim(0, configuration['to_wind_speed'])
    plt.xlabel(x_axis_label)
    plt.ylabel("Boat Speed Residuals [kn]")
    frame_residual.plot(X_scaler.inverse_transform(X_test[:, 0].reshape(-1, 1)),
                        y_scaler.inverse_transform(differences), 'rx')
    plt.grid()


def __add_to_plot(X_scaler, label, x_plot, y_plot, y_scaler):
    plt.plot(X_scaler.inverse_transform(x_plot.reshape(-1, 1)), y_scaler.inverse_transform(y_plot.reshape(-1, 1)),
             label=label)


def __prepare_plot(X_scaler, configuration, estimators, this_X_test, this_y_test, title, x_axis_label, y_scaler):
    fig = plt.figure(figsize=(10, 8))
    if len(estimators) == 1:
        fig.add_axes((.1, .35, .8, .55))
    plt.xlim(0, configuration['to_wind_speed'])
    plt.xlabel(x_axis_label)
    plt.ylim(0, configuration['to_boat_speed'])
    plt.ylabel("Boat Speed [kn]")
    if configuration['show_title_in_plot']:
        plt.title(title)
    plt.plot(X_scaler.inverse_transform(this_X_test[:, 0].reshape(-1, 1)), y_scaler.inverse_transform(this_y_test),
             'k+')  # test set
    plt.grid()
    return fig
```

Listing A.18: data˙analysis/fit3D.py

```
"""
Performs data analysis of true wind speed, true wind angle and boat speed.

Uses the extendable interface of scikit-learn to fit 3-dimensional models to the data.
Loads data from the database and performs pre-processing on the data.
Calculates weights of the data to consider influence on the result according to the quality of single data points.
Supports various types of estimators for the optimization problem.
Generates plots from the result.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging
import timeit
from math import sqrt
from random import randrange

import numpy
import scipy
from matplotlib import animation
from matplotlib import colors
from matplotlib import pyplot as plt, rc_context
from matplotlib.collections import PolyCollection
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from sklearn.cross_validation import train_test_split
```

```python
from sklearn.externals import joblib

from polar_diagram_generator.converter import wind_converter
from polar_diagram_generator.data_analysis import weight_calculation, polar_diagram_plotter
from polar_diagram_generator.data_analysis.estimator.odr_estimator import ODREstimator
from polar_diagram_generator.data_analysis.models import models2d
from polar_diagram_generator.data_analysis.models import models3d
from polar_diagram_generator.data_analysis.models.models3d import DynamicCallableModel
from polar_diagram_generator.exploratory_data_analysis import speed_trend
from polar_diagram_generator.util.strings import TWA_TICK_LABELS_WITH_NAME

logger = logging.getLogger('polar.data_analysis.fit')


RC_DEFINITION = {}
LINE_COLORS = [colors.rgb2hex([0.0, 0.0, 0.5]),
               colors.rgb2hex([0.0, 0.0, 1]),
               colors.rgb2hex([0.0, 0.38, 1]),
               colors.rgb2hex([0.0, 0.83, 1]),
               colors.rgb2hex([0.3, 1.0, 0.67]),
               colors.rgb2hex([1.0, 0.48, 0]),
               colors.rgb2hex([1.0, 0.07, 0]),
               colors.rgb2hex([0.5, 0.0, 0])
               ]

PLOT_3D_SPLINE_GRID = False
PLOT_3D_SURFACE = False
PLOT_3D_FIT = True
PLOT_POLAR_FIT = True
SHOW_HISTOGRAMS = False
SHOW_WHOLE_LINES = False
SHOW_BOTTOM_RIGHT_INFO_TEXT = False
PLOT_WEIGHT_TREND = True
MAX_ERROR = 1
PREDICTION_PROBABILITY = 0.10


def __load_3d_values(cursor, configuration):
    parameters = (configuration['from_wind_angle'],
                  configuration['to_wind_angle'],
                  configuration['from_timestamp'],
                  configuration['to_timestamp'],
                  configuration['from_mark'],
                  configuration['to_mark'])
    cursor.execute('SELECT mark, tws, twa, bsp FROM logger'
                   ' WHERE bsp IS NOT NULL'
                   ' AND twa IS NOT NULL'
                   ' AND tws IS NOT NULL'
                   ' AND twa >= ?'
                   ' AND twa <= ?'
                   ' AND outlier IS NOT 42'
                   ' AND ((timestamp > ? AND timestamp <= ?) OR timestamp IS NULL)'
                   ' AND ((mark > ? AND mark <= ?) OR mark IS NULL)'
                   , parameters)
    features = cursor.fetchall()
    return features


def load_3d_values_with_tuples(cursor, configuration):
    """
    Loads data from the database.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    :return: measured data as list of tuples
    """
    if configuration['mark_tuples'] == None or len(configuration['mark_tuples']) <= 0:
        raise RuntimeError("mark_tuples not initialized or empty")

    parameters = (configuration['from_wind_angle'],
                  configuration['to_wind_angle'],
                  configuration['from_wind_speed'],
                  configuration['to_wind_speed'],
                  configuration['from_boat_speed'])
    query = 'SELECT mark, tws, twa, bsp FROM logger' \
            ' WHERE bsp IS NOT NULL' \
            ' AND twa IS NOT NULL' \
            ' AND tws IS NOT NULL' \
            ' AND twa >= ?' \
            ' AND twa < ?' \
            ' AND tws >= ?' \
            ' AND tws < ?' \
            ' AND bsp >= ?' \
            ' AND outlier IS NOT 42' \
            ' AND ('
    first = True
    for start, end in configuration['mark_tuples']:
        if not first:
            query += ' OR '
        first = False
        query += ' mark BETWEEN ' + str(start) + ' AND ' + str(end) + ' '

    query += ' )'
    cursor.execute(query, parameters)
    features = cursor.fetchall()
    return features


def perform(cursor, configuration):
    """
    Performs data analysis of true wind speed, true wind angle and boat speed.
    :param cursor: database cursor with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    if len(configuration['mark_tuples']) > 0:
        features = numpy.asarray(load_3d_values_with_tuples(cursor, configuration))
    else:
        features = numpy.asarray(__load_3d_values(cursor, configuration))

    configuration['from_wind_angle'] = 0
    configuration['to_wind_angle'] = 180
    configuration['from_boat_speed'] = 0
    configuration['to_boat_speed'] = 10

    mark_list = features[:, 0]
    X_list = features[:, 1:3]
    y_list = features[:, 3]

    mark = numpy.asarray(mark_list)
    X = numpy.asarray(X_list)
    y = numpy.asarray(y_list)

    random_state = randrange(1000)
    logger.debug("random_state: %d" % random_state)

    weights_bsp, weights_tws, weights_twa = weight_calculation.calculate_weights(X, y, configuration=configuration)

    if PLOT_WEIGHT_TREND:
        speed_trend.plot(mark, y_list, weights_tws, "Boat speed trend with belonging weights",
                         file_path=configuration['output_path'] + "plots/weights_trend_bsp.png",
                         y_label="Weights           Speed [knots]", legend=["Boat Speed", "Boat Speed weights"])

    X_train, X_test, y_train, y_test, weights_bsp_train, weights_bsp_test, weights_tws_train, weights_tws_test,
         weights_twa_train, weights_twa_test = train_test_split(
        X, y, weights_bsp, weights_tws, weights_twa, test_size=0.3, random_state=random_state)

    __fit_model(X_train, X_test, y_train, y_test, weights_bsp_train, weights_bsp_test, weights_tws_train,
                weights_tws_test, weights_twa_train, weights_twa_test, configuration=configuration)


def __fit_model(X_train, X_test, y_train, y_test, weights_bsp_train, weights_bsp_test, weights_tws_train,
                weights_tws_test, weights_twa_train, weights_twa_test, configuration):
    joblib.dump(X_train, configuration['output_path'] + 'models/data_X_train.np')
    joblib.dump(X_test, configuration['output_path'] + 'models/data_X_test.np')
    joblib.dump(y_train, configuration['output_path'] + 'models/data_y_train.np')
    joblib.dump(y_test, configuration['output_path'] + 'models/data_y_test.np')
    latex_output = ""

    weights_bsp, weights_tws, weights_twa = weights_bsp_train, weights_tws_train, weights_twa_train
    uniform_weight = numpy.ones(len(weights_bsp_train))

    weight_tuples = [
        ("uniform", uniform_weight, uniform_weight, uniform_weight),
        ("weighted", weights_bsp, weights_tws, weights_twa)
    ]

    model_definitions = []

    if configuration['use_model_3d_tws_twa_concave'] is True:
        model_definitions.append(('ODR Concave with Downturn for Wind Speed and Angle', 'Concave with Downturn \\newline for
             Wind Speed and Angle (\\ref{eq:3d-hypothesis-concave-with-downturn})', ODREstimator(models3d.
             model_tws_concave_with_downturn_and_twa, [1.0, 1.0, 1.0, 90.0, 20.0, 0.0, 0.0], (uniform_weight, uniform_weight),
             uniform_weight), models3d.model_tws_concave_with_downturn_and_twa, models3d.
             derivatives_model_tws_concave_with_downturn_and_twa))
    if configuration['use_model_3d_tws_twa_concave_weighted'] is True:
        model_definitions.append(('ODR␣Concave␣with␣Downturn␣for␣Wind␣Speed␣and␣Angle␣weighted', 'weighted␣Concave␣with␣
             Downturn␣\\newline␣for␣Wind␣Speed␣and␣Angle␣(\\ref{eq:3d-hypothesis-concave-with-downturn})', ODREstimator(
             models3d.model_tws_concave_with_downturn_and_twa, [1.0, 1.0, 1.0, 90.0, 20.0, 0.0, 0.0], (weights_tws, weights_twa
             ), weights_bsp), models3d.model_tws_concave_with_downturn_and_twa, models3d.
             derivatives_model_tws_concave_with_downturn_and_twa))
    if configuration['use_model_3d_tws_twa_s_shaped'] is True:
        model_definitions.append(('ODR␣S-shaped␣with␣Saturation␣Limit␣and␣Downturn␣for␣Wind␣Speed␣and␣Angle', 'S-shaped␣with␣
             Saturation␣Limit␣and␣Downturn␣\\newline␣for␣Wind␣Speed␣and␣Angle␣(\\ref{eq:3d-hypothesis-both-s-shaped})',
             ODREstimator(models3d.model_tws_and_twa_s_shaped_with_downturn, [5.0, 8.0, 10.0, 1.0, 4.0, 15.0, 1.0, 1.0, 1.0], (
             uniform_weight, uniform_weight), uniform_weight), models3d.model_tws_and_twa_s_shaped_with_downturn, models3d.
             derivatives_model_tws_and_twa_s_shaped_with_downturn))
    if configuration['use_model_3d_tws_twa_s_shaped_weighted'] is True:
        model_definitions.append(('ODR␣S-shaped␣with␣Saturation␣Limit␣and␣Downturn␣for␣Wind␣Speed␣and␣Angle␣weighted', '
             weighted␣S-shaped␣with␣Saturation␣Limit␣and␣Downturn␣\\newline␣for␣Wind␣Speed␣and␣Angle␣(\\ref{eq:3d-hypothesis-
             both-s-shaped})', ODREstimator(models3d.model_tws_and_twa_s_shaped_with_downturn, [5.0, 8.0, 10.0, 1.0, 4.0, 15.0,
              1.0, 1.0, 1.0], (weights_tws, weights_twa), weights_bsp), models3d.model_tws_and_twa_s_shaped_with_downturn,
             models3d.derivatives_model_tws_and_twa_s_shaped_with_downturn))
    if configuration['use_model_tws_3d_saturation_limit_twa_gauss'] is True:
        model_definitions.append(('ODR␣S-shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣Gaussian␣model␣for␣Wind␣Angle', 'S
             -shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣\\newline␣Gaussian␣model␣for␣Wind␣Angle␣(\\ref{eq:3d-
             hypothesis})', ODREstimator(models3d.model_tws_s_shaped_with_downturn_and_twa_gaussian, [1.0, 1.0, 1.0, 90.0, 4.0,
              15.0, 1.0], (uniform_weight, uniform_weight), uniform_weight), models3d.
             model_tws_s_shaped_with_downturn_and_twa_gaussian, models3d.
             derivatives_model_tws_s_shaped_with_downturn_and_twa_gaussian))
    if configuration['use_model_tws_3d_saturation_limit_twa_gauss_weighted'] is True:
        model_definitions.append(('ODR␣S-shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣Gaussian␣model␣for␣Wind␣Angle␣
             weighted', 'weighted␣S-shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣\\newline␣Gaussian␣model␣for␣Wind␣Angle
             ␣(\\ref{eq:3d-hypothesis})', ODREstimator(models3d.model_tws_s_shaped_with_downturn_and_twa_gaussian, [1.0, 1.0,
             1.0, 90.0, 4.0, 15.0, 1.0], (weights_tws, weights_twa), weights_bsp), models3d.
             model_tws_s_shaped_with_downturn_and_twa_gaussian, models3d.
             derivatives_model_tws_s_shaped_with_downturn_and_twa_gaussian))
    if configuration['use_model_tws_3d_saturation_limit_downturn_twa_gauss'] is True:
        model_definitions.append(('ODR␣S-shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣Gaussian␣model␣for␣Wind␣Angle␣+␣
             Combination', 'S-shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣\\newline␣Gaussian␣model␣for␣Wind␣Angle␣+␣
             Combination␣(\\ref{eq:3d-hypothesis-extended})', ODREstimator(models3d.
             model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination, [1.0, 1.0, 1.0, 4.0, 15.0, 1.0, 1.0, 1.0], (
             uniform_weight, uniform_weight), uniform_weight), models3d.
             model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination, models3d.
             derivatives_model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination))
    if configuration['use_model_tws_3d_saturation_limit_downturn_twa_gauss_weighted'] is True:
        model_definitions.append(('ODR␣S-shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣Gaussian␣model␣for␣Wind␣Angle␣+␣
             Combination␣weighted', 'weighted␣S-shaped␣with␣Sat.␣Limit␣and␣Downturn␣for␣Wind␣Speed;␣\\newline␣Gaussian␣model␣
             for␣Wind␣Angle␣+␣Combination␣(\\ref{eq:3d-hypothesis-extended})', ODREstimator(models3d.
             model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination, [1.0, 1.0, 1.0, 4.0, 15.0, 1.0, 1.0, 1.0], (
             weights_tws, weights_twa), weights_bsp), models3d.
             model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination, models3d.
             derivatives_model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination))

    if len(model_definitions) == 0:
         __add_all_combinations_to_model_definitions(model_definitions, weight_tuples)

    is_scatter_plot_without_fit = len(model_definitions) == 0

    for title, this_X, this_y, this_X_test, this_y_test in [('Polar␣diagram␣fit', X_train, y_train, X_test, y_test)]:

        for name, latex_name, estimator, model, model_partial_derivatives in model_definitions:
             logger.debug("\n\n\n================================================\nEstimator:␣%s" % name)

             if PLOT_3D_FIT:
                ax, fig = __plot_3d_fit(configuration, is_scatter_plot_without_fit, this_X_test, this_y_test)

             start = timeit.default_timer()
             estimator.fit(this_X, this_y)
             stop = timeit.default_timer()
             runtime_seconds = (stop - start)
             logger.debug('Runtime:␣' + str(runtime_seconds))
             if 'ODR' not in name:
                joblib.dump(estimator, configuration['output_path'] + 'models/model_' + name + '.pkl')

             sum_square = estimator.sum_square
             text = ''

             uniform_weight_test = numpy.ones(len(this_y_test))

             logger.debug("\n\n\n\n\n--------------------␣TEST␣DATA␣UNIFORM␣--------------------")
             odr_estimator_test_data = ODREstimator(
                model, estimator.popt,
                (uniform_weight_test, uniform_weight_test), uniform_weight_test, max_iterations=0)
             odr_estimator_test_data.fit(this_X_test, this_y_test)

             sum_square_test_uniform = odr_estimator_test_data.output.sum_square
             logger.debug("------------------------------------------------------------")
             logger.debug("\n\n\n\n\n--------------------- TEST DATA WEIGHTED -----------------------")
             odr_estimator_test_data = ODREstimator(
                model, estimator.popt,
                (weights_tws_test, weights_twa_test), weights_bsp_test, max_iterations=0)
             odr_estimator_test_data.fit(this_X_test, this_y_test)

             sum_square_test_weighted = odr_estimator_test_data.output.sum_square
             logger.debug("------------------------------------------------------------")

             if PLOT_3D_SPLINE_GRID:
                __plot_3d_spline_grid(ax, X_train, y_train)

             if PLOT_3D_SURFACE:
                tws_axis = numpy.arange(4, 22, 1)
                twa_axis = numpy.arange(25, 181, 3)
                X_mesh_test = numpy.asarray((numpy.ravel(tws_axis), numpy.ravel(twa_axis))).T
                filename = configuration['output_path'] + 'models/errors_mesh_' + name + '.pkl'
                errors = numpy.asarray(__calculate_errors(X_mesh_test, X_train, None, None))
                joblib.dump(errors, filename)

                prediction = estimator.predict(X_mesh_test)
                __plot_3d_surface(ax, prediction, errors, tws_axis, twa_axis)

             if PLOT_3D_FIT:
                __plot_3d_fit_and_animations(X_train, ax, configuration, estimator, fig, name, text, title)

             if PLOT_POLAR_FIT:
                ax, fig, title, start_end_tuples = __plot_polar_fit(X_train, ax, configuration, estimator, fig, name, title)
                polar_fit = polar_diagram_plotter.PolarFit(vpp=None, estimator=estimator, start_end_tuples=start_end_tuples)
                polar_diagram_plotter.plot_polar_fit(data=polar_fit, configuration=configuration, name=name)

             latex_output += ("%s & %.2f~s & %.2f & %.2f & %.2f \\\\\n" % (
             latex_name, runtime_seconds, sum_square, sum_square_test_uniform, sum_square_test_weighted))

        if PLOT_3D_FIT and is_scatter_plot_without_fit:
             __add_ticks_to_plot()
             if configuration['show_plots'] is True:
                plt.show()

    logger.debug("\n\n\n============== Summary ===============\nNumber of estimators in this run: %d" % len(model_definitions))
    for name, latex_name, estimator, model, model_partial_derivatives in model_definitions:
        logger.debug("- %s" % name)

    header = "%s & %s & %s & %s & %s" % (
    "Name", "Runtime [s]", "\chi^2_{train}", "\chi^2_{test uniform}", "\chi^2_{test weighted}")
    logger.debug("\n---- LaTeX Result Table ----\n%s\n%s" % (header, latex_output))


def __add_all_combinations_to_model_definitions(model_definitions, weight_tuples):
    for (name_tws, model_tws) in models2d.all_models():
        for (name_twa, model_twa) in models2d.all_models():
             for (name_combination, model_combination) in models2d.all_combination_models():
                for (name_weighting, this_weights_bsp, this_weights_tws, this_weights_twa) in weight_tuples:
                    dynamic_callable_model = DynamicCallableModel(model_tws, model_twa, model_combination)
                    model_name = 'ODR - %s - %s - %s - %s' % (name_tws, name_twa, name_combination, name_weighting)
                    model_latex_output = '%s & %s & %s & %s' % (name_tws, name_twa, name_combination, name_weighting)
                    odr_estimator = ODREstimator(dynamic_callable_model, dynamic_callable_model.initial_parameters(),
                                                 (this_weights_tws, this_weights_twa), this_weights_bsp)
                    model_definitions.append((model_name, model_latex_output, odr_estimator, dynamic_callable_model,
                                              models3d.derivatives_model_tws_s_shaped_with_downturn_and_twa_gaussian))


#####################################################

def __add_ticks_to_plot():
    ticks = range(-180, 181, 45)
    plt.yticks(ticks, ticks)


def __plot_polar_fit(X_train, ax, configuration, estimator, fig, name, title):
    with rc_context(RC_DEFINITION):
        start_end_tuples = []

        fig = plt.figure(figsize=(15, 10))
        ax = plt.subplot(111, projection='polar')
        title = 'Polar␣Diagram␣␣−␣%s' % name
        ax.set_title('%s' % title, va='bottom')
        ax.set_theta_direction(-1)
        ax.set_theta_offset(numpy.pi / 2.0)
        ax.set_xticklabels(TWA_TICK_LABELS_WITH_NAME)  # Twa
        ax.set_yticklabels([2, 4, 6, 8, '10␣Boat␣Speed␣[kn]'])  # Bsp
        ax.set_ylim(0, 10)

        for index, speed in enumerate([6., 8., 10., 12., 14., 16., 20.]):
            x_plot = None
            for angle in range(25, 180, 7):
                if x_plot is None:
                    x_plot = [[speed, angle]]
                else:
                    x_plot = numpy.concatenate((x_plot, [[speed, angle]]), axis=0)

            err = numpy.asarray(__calculate_errors(x_plot, X_train, None, None))

            start, end = __calculate_start_and_end(err)
            if SHOW_WHOLE_LINES:  # DEBUG show whole line
                start = 0
                end = len(err)
            start_end_tuples.append((speed, x_plot[start][1], x_plot[end][1]))

            err = err[start:end]

            x_plot = x_plot[start:end]

            if len(x_plot) == 0:
                logger.debug("Skipped␣polar␣line␣for␣speed␣" + str(speed))
                continue

            __polar_plot(ax, x_plot, estimator.predict(x_plot), err, 'Fit␣at␣%d␣kn' % (speed), LINE_COLORS[index])

        __add_orc_data_to_plot(ax, line_colors=LINE_COLORS)

        ax.axis([0, 2 * numpy.pi, 0, 11])

        plt.legend(loc=2, frameon=False, bbox_to_anchor=(1.05, 1),
                   title='True␣Wind␣Speed:')
        plt.draw()
        plt.savefig(configuration['output_path'] + 'plots/fit_3d_polar_' + name + '.png')
        if configuration['show_plots'] is True:
            plt.show()
    return ax, fig, title, start_end_tuples


def __plot_3d_fit_and_animations(X_train, ax, configuration, estimator, fig, name, text, title):
    for index, speed in enumerate([6., 8., 10., 12., 14., 16., 20.]):
        x_plot = None
        for angle in range(25, 180, 7):
            if x_plot is None:
                x_plot = [[speed, angle]]
            else:
                x_plot = numpy.concatenate((x_plot, [[speed, angle]]), axis=0)

        err = numpy.asarray(__calculate_errors(x_plot, X_train, None, None))

        start, end = __calculate_start_and_end(err)
        if SHOW_WHOLE_LINES:  # DEBUG show whole line
            start = 0
            end = len(err)
        err = err[start:end]

        x_plot = x_plot[start:end]

        if len(x_plot) == 0:
            logger.debug("Skipped␣line␣for␣speed␣" + str(speed))
            continue

        y_plot = estimator.predict(x_plot)

        ax.plot(x_plot[:, 0], x_plot[:, 1], zs=y_plot, label='%s␣%dkn' % (name, speed), color=LINE_COLORS[index])

        upperband = y_plot + err
        if PLOT_3D_SURFACE:
            lowerband = y_plot
        else:
            lowerband = y_plot - err
        verts_pred = list(zip(x_plot[:, 0], x_plot[:, 1], lowerband)) + list(
            zip(x_plot[:, 0][::-1], x_plot[:, 1][::-1], upperband[::-1]))
        poly = Poly3DCollection([verts_pred], closed=True, alpha=0.3,
                                label="PI_(%g)" % PREDICTION_PROBABILITY)
        poly.set_facecolor(LINE_COLORS[index])
    if SHOW_BOTTOM_RIGHT_INFO_TEXT:
        fig.text(0.95, 0.02, text, color='k', fontsize=7, ha='right', transform=ax.transAxes)
    plt.title(title)

    ax.view_init(25, 195)

    plt.savefig(configuration['output_path'] + "plots/fit_3d_" + name + ".png")

    if configuration['perform_animation_plots']:
        ani = animation.FuncAnimation(fig, __animate, 360, interval=100, blit=False,
                                      fargs=(ax,))  # 360 is the number of frames
        ani.save(configuration['output_path'] + 'animation/3d-fit.mp4', fps=15)

    if configuration['show_plots'] is True:
        plt.show()


def __plot_3d_surface(ax, prediction, errors, tws_axis, twa_axis):
    tws_axis, twa_axis = numpy.meshgrid(tws_axis, twa_axis)
    shape = tws_axis.shape
    for index, error in enumerate(errors):
        if error >= MAX_ERROR:
            prediction[index] = numpy.nan
    surface_fit = ax.plot_surface(tws_axis, twa_axis, prediction.reshape(shape), cmap=plt.cm.plasma, vmin=0.0,
                                  vmax=10.0, alpha=0.9)


def __plot_3d_spline_grid(ax, X_train, y_train):
    X = X_train[:, 0]
    Y = X_train[:, 1]
    Z = y_train
    xi = numpy.linspace(5, 21, 15)
    yi = numpy.linspace(30, 121, 10)
    zi = scipy.interpolate.griddata((X, Y), Z, (xi[None, :], yi[:, None]), method='cubic')
    xig, yig = numpy.meshgrid(xi, yi)
    surface_spline = ax.plot_surface(xig, yig, zi,
                                     linewidth=0, cmap=plt.cm.YlOrBr, vmin=0.0, vmax=10.0, alpha=1.0)


def __plot_3d_fit(configuration, is_scatter_plot_without_fit, this_X_test, this_y_test):
    fig = plt.figure(figsize=(20, 10))
    ax = Axes3D(fig)
    ax.set_xlim3d(configuration['from_wind_speed'], configuration['to_wind_speed'])
    ax.set_xlabel("True_Wind_Speed")
    ax.set_ylim3d(configuration['from_wind_angle'], configuration['to_wind_angle'])
    ax.set_ylabel("True_Wind_Angle")
    ax.set_zlim3d(configuration['from_boat_speed'], configuration['to_boat_speed'])
    ax.set_zlabel("Boat_Speed")
    if is_scatter_plot_without_fit:
        ax.view_init(15, 170)
    ax.scatter(this_X_test[:, 0], this_X_test[:, 1], this_y_test, marker='+', c=this_y_test, cmap=plt.cm.coolwarm,
               alpha=1.0)  # test set
    return ax, fig


def __animate(i, axis):
    logger.debug("animate_%f" % i)
    axis.azim = i
    axis.elev = -52 - (-abs(i - 180) / 2)
    plt.draw()
    return axis


def __polar_plot(axes, x_plot, z, weights, label, linecolor):
    rho = wind_converter.wind_angle_cart_2_pol(x_plot[:, 1])
    phi = z

    axes.plot(rho, phi, label=label, color=linecolor)
    upperband = z + weights
    lowerband = z - weights
    for index, value in enumerate(upperband):
        if value < 0:
            upperband[index] = 0
    for index, value in enumerate(lowerband):
        if value < 0:
            lowerband[index] = 0

    verts_pred = list(zip(rho, lowerband)) + list(
        zip(rho[::-1], upperband[::-1]))
    poly = PolyCollection([verts_pred], closed=True, alpha=0.3,
                          label="PI")
    poly.set_facecolor(linecolor)
    axes.add_collection(poly)


def __add_orc_data_to_plot(ax, line_colors):
    other_angles = [52, 60, 70, 75, 80, 90, 110, 120, 135, 150, 165, 180]

    velocities = [(6, 44.2, [4.61, 5.1, 5.42, 5.64, 5.69, 5.71, 5.62, 4.96, 4.42, 3.65, 3.25, 3.04, 2.95]),
                  (8, 42.7, [5.45, 6.09, 6.43, 6.67, 6.72, 6.73, 6.64, 5.93, 5.43, 4.70, 4.26, 4.01, 3.91]),
                  (10, 42.3, [6.17, 6.91, 7.22, 7.40, 7.43, 7.44, 7.37, 6.74, 6.25, 5.57, 5.16, 4.90, 4.79]),
                  (12, 42.1, [6.79, 7.45, 7.67, 7.79, 7.81, 7.82, 7.76, 7.36, 6.97, 6.33, 5.94, 5.69, 5.57]),
                  (14, 40.7, [6.98, 7.69, 7.91, 8.05, 8.08, 8.08, 8.03, 7.73, 7.48, 7.01, 6.65, 6.40, 6.28]),
                  (16, 39.6, [7.06, 7.79, 8.01, 8.23, 8.29, 8.32, 8.28, 7.99, 7.81, 7.51, 7.26, 7.05, 6.94]),
                  (20, 39.6, [7.20, 7.87, 8.1, 8.38, 8.52, 8.66, 8.78, 8.49, 8.31, 8.11, 7.98, 7.88, 7.82])]
    other_angles_polar = wind_converter.wind_angle_cart_2_pol(numpy.asarray(other_angles))

    for index, velocity in enumerate(velocities):
        speed, best_angle, boat_speeds = velocity
        best_angle_polar = wind_converter.wind_angle_cart_2_pol(best_angle)
        ax.scatter(numpy.append([best_angle_polar], other_angles_polar, axis=0), boat_speeds, color=line_colors[index],
                   label="VPP_at_%d_kn" % speed)


def __calculate_start_and_end(err):
    start = 0
    end = 0
    for index, error in enumerate(err):
        if not (error == 0.0 or error >= MAX_ERROR):
            start = index
            break
    for index, error in reversed(list(enumerate(err))):
        if not (error == 0.0 or error >= MAX_ERROR):
            end = index
            break
    return start, end


def __calculate_errors(x_plot, X_train, predictions, model):
    weights = []
    for index, x in enumerate(x_plot):
        trainingElementsInRange = [i for i in X_train if
                                   i[0] >= (x[0] - 3) and i[0] < (x[0] + 3) and i[1] >= (x[1] - 1.5) and i[1] < (
                                   x[1] + 1.5)]
        if len(trainingElementsInRange) == 0:
            weights.append(0)
        else:
            weights.append(10 / sqrt(len(trainingElementsInRange)))
    for index, weight in enumerate(weights):
        if weight == 0 or weight > MAX_ERROR:
            weights[index] = MAX_ERROR

    return weights
```

Listing A.19: data_analysis/noise_filtering.py

```
"""
Performs calculation of the smoothness of true wind speed, true wind angle and boat speed.

Plots the differences in the consecutive values of each feature and calculates a confidence interval.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import matplotlib.pyplot as plt
import numpy

import polar_diagram_generator.converter.wind_converter as wind_converter

CONFIDENCE_INTERVAL_INT = 99
```

```python
def __load_values(cursor, configuration):
    parameters = (configuration['from_timestamp'], configuration['to_timestamp'])
    cursor.execute('SELECT mark, bsp, twa, tws FROM logger '
                   ' WHERE bsp IS NOT NULL'
                   ' AND twa IS NOT NULL'
                   ' AND tws IS NOT NULL'
                   ' AND ((timestamp >= ? AND timestamp <= ?) OR timestamp IS NULL)'
                   , parameters)
    val = cursor.fetchall()
    return val


def __plot(differences_array, upper, lower, title, unit, kernel_size, file_path, show_title=True):
    plt.figure()
    if show_title:
        plt.title("Moving Average Deviation Histogram of " + title)
    bins = numpy.linspace(-1, 1, 0.1)
    plt.hist(differences_array, bins=1000, log=True)
    plt.xlabel("difference of an element to the mean of the previous " + str(kernel_size) + " elements [" + unit + "]")
    plt.ylabel("Frequency")
    label = str(CONFIDENCE_INTERVAL_INT) + "% confidence interval\n" + str(lower) + " to " + str(upper)
    lower_line = plt.axvline(lower, label=label, color="r", ls="--")
    plt.axvline(upper, color="r", ls="--")
    plt.legend(handles=[lower_line])
    plt.grid(True)
    plt.draw()
    plt.savefig(file_path)


def perform(db_handler, configuration):
    """
    Performs calculation of the smoothness of true wind speed, true wind angle and boat speed.
    :param db_handler: database handle with a valid table 'logger'
    :param configuration: configuration array to control data range and outputs
    """
    __print_twa(db_handler, configuration)


def __print_twa(db_handler, configuration):
    differences_bsp = []
    differences_twa = []
    differences_tws = []
    differences = []
    count_result = __load_values(db_handler.get_cursor(), configuration)

    window_size = 5
    for index, entry in enumerate(count_result):
        if index >= window_size:
            sum_bsp = 0
            sum_twa_0_to_360 = 0
            sum_twa_180_to_180 = 0
            sum_tws = 0
            for i in range(-window_size+1, 1):
                sum_bsp += count_result[index + i][1]
                sum_twa_0_to_360 += wind_converter.wind_angle_convert_to_range_0_to_360(count_result[index + i][2])
                sum_twa_180_to_180 += wind_converter.wind_angle_convert_to_range_180_to_180(count_result[index + i][2])
                sum_tws += count_result[index + i][3]
            differences_bsp.append(sum_bsp / window_size - entry[1])
            diff_0_to_360 = wind_converter.wind_angle_convert_to_range_180_to_180(sum_twa_0_to_360 / window_size -
                wind_converter.wind_angle_convert_to_range_0_to_360(entry[2]))
            diff_180_to_180 = wind_converter.wind_angle_convert_to_range_180_to_180(sum_twa_180_to_180 / window_size -
                wind_converter.wind_angle_convert_to_range_180_to_180(entry[2]))
            if abs(diff_0_to_360) < abs(diff_180_to_180):
                diff_twa = diff_0_to_360
            else:
                diff_twa = diff_180_to_180
            differences_twa.append(diff_twa)
            differences_tws.append(sum_tws / window_size - entry[3])
            differences.append((entry[0], sum_bsp / window_size - entry[1], diff_twa, diff_0_to_360, sum_tws / window_size -
                entry[3]))

    quantiles = __calculate_quantiles()
    lower_bsp, upper_bsp = numpy.percentile(differences_bsp, quantiles)
    __plot(differences_bsp, upper_bsp, lower_bsp, "Boat Speed", "kn", window_size,
           configuration['output_path'] + "plots/moving_average_deviation_bsp.png", show_title=configuration['
               show_title_in_plot'])

    lower_twa, upper_twa = numpy.percentile(differences_twa, quantiles)
    __plot(differences_twa, upper_twa, lower_twa, "True_Wind_Angle", "degree", window_size,
           configuration['output_path'] + "plots/moving_average_deviation_twa.png", show_title=configuration['
                show_title_in_plot'])

    lower_tws, upper_tws = numpy.percentile(differences_tws, quantiles)
    __plot(differences_tws, upper_tws, lower_tws, "True_Wind_Speed", "kn", window_size,
           configuration['output_path'] + "plots/moving_average_deviation_tws.png", show_title=configuration['
                show_title_in_plot'])

    outliers = []
    db_handler.get_cursor().execute('UPDATE_logger_SET_outlier_=_0_WHERE_outlier_IS_1')
    for difference in differences:
        if not (lower_bsp < difference[1] < upper_bsp and lower_twa < difference[2] < upper_twa and lower_tws < difference[4] <
                upper_tws):
            db_handler.get_cursor().execute('UPDATE_logger_SET_outlier_=_1_WHERE_mark_IS_?', (difference[0],))
            outliers.append(difference[0])

    db_handler.commit_transaction()

    plt.show()


def __calculate_quantiles():
    diff = (100.0 - float(CONFIDENCE_INTERVAL_INT)) / 2.0
    quantiles = [0.0 + diff, 100 - diff]
    return quantiles
```

Listing A.20: data˙analysis/weight˙calculation.py

```
"""
Performs weight calculation of true wind speed, true wind angle and boat speed.

Assumes that values that follow an homogeneous sequence contribute more to the best fitted model.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0

import numpy
from matplotlib import pyplot as plt


def calculate_weights(X_train, y_train, configuration):
    """
    Performs weight calculation of true wind speed, true wind angle and boat speed.
    :param X_train: list with the training data set's independent values. Tuple with TWS at index 0 and TWA at index 1
    :param y_train: list with the training data set's dependent value. Represents the BSP.
    :param configuration: specifies if histograms should be made (plot_weight_histograms) and the output path for plots (
           output_path)
    :return: Tuple with standard deviation for BSP, TWS and TWA with the same length as input list
    """
    return __calculate_weights_continuously(X_train, y_train, configuration)


def __calculate_weights_binary(X_train, y_train, configuration):
    weights_bsp, weights_tws, weights_twa = __calculate_standard_deviations(X_train, y_train)
    lower_bsp, upper_bsp = numpy.percentile(weights_bsp, [0.0, 95.0])
    lower_tws, upper_tws = numpy.percentile(weights_tws, [0.0, 95.0])
    lower_twa, upper_twa = numpy.percentile(weights_twa, [0.0, 95.0])
    __calculate_binary_weight(weights_bsp, lower_limit=lower_bsp, upper_limit=upper_bsp)
    __calculate_binary_weight(weights_tws, lower_limit=lower_tws, upper_limit=upper_tws)
    __calculate_binary_weight(weights_twa, lower_limit=lower_twa, upper_limit=upper_twa)

    if configuration['plot_weight_histograms'] is True:
        __plot_histogram(weights_bsp, 'Histogram_of_normalized_boat_speed_weights', filepath=configuration['output_path'] + "
             plots/weights_for_fit_3d_bsp.png", show_plot=configuration['show_weight_histograms'] is True)
        __plot_histogram(weights_tws, 'Histogram_of_normalized_true_wind_speed_weights', filepath=configuration['output_path']
             + "plots/weights_for_fit_3d_tws.png", show_plot=configuration['show_weight_histograms'] is True)
        __plot_histogram(weights_twa, 'Histogram_of_normalized_true_wind_angle_weights', filepath=configuration['output_path']
             + "plots/weights_for_fit_3d_twa.png", show_plot=configuration['show_weight_histograms'] is True)

    return weights_bsp, weights_tws, weights_twa


def __calculate_weights_continuously(X_train, y_train, configuration):
    weights_bsp, weights_tws, weights_twa = __calculate_standard_deviations(X_train, y_train)
    lower_bsp, upper_bsp = numpy.percentile(weights_bsp, [0.1, 95.0])
    lower_tws, upper_tws = numpy.percentile(weights_tws, [0.1, 95.0])
    lower_twa, upper_twa = numpy.percentile(weights_twa, [0.1, 95.0])

    if configuration['plot_weight_histograms'] is True:
        __plot_histogram(weights_bsp, 'Histogram_of_boat_speed_standard_deviations', filepath=configuration['output_path'] + "
            plots/standard_deviations_for_fit_3d_bsp.png", show_plot=configuration['show_weight_histograms'] is True)
        __plot_histogram(weights_tws, 'Histogram_of_true_wind_speed_standard_deviations', filepath=configuration['output_path']
             + "plots/standard_deviations_for_fit_3d_tws.png", show_plot=configuration['show_weight_histograms'] is True)
        __plot_histogram(weights_twa, 'Histogram_of_true_wind_angle_standard_deviations', filepath=configuration['output_path']
             + "plots/standard_deviations_for_fit_3d_twa.png", show_plot=configuration['show_weight_histograms'] is True)

    __invert_and_limit(weights_bsp, lower_limit=lower_bsp, upper_limit=upper_bsp)
    __invert_and_limit(weights_tws, lower_limit=lower_tws, upper_limit=upper_tws)
    __invert_and_limit(weights_twa, lower_limit=lower_twa, upper_limit=upper_twa)

    weights_bsp_scaled = weights_bsp / numpy.mean(weights_bsp)
    weights_tws_scaled = weights_tws / numpy.mean(weights_tws)
    weights_twa_scaled = weights_twa / numpy.mean(weights_twa)

    if configuration['plot_weight_histograms'] is True:
        __plot_histogram(weights_bsp_scaled, 'Histogram_of_normalized_boat_speed_weights', filepath=configuration['output_path'
            ] + "plots/weights_for_fit_3d_bsp.png")
        __plot_histogram(weights_tws_scaled, 'Histogram_of_normalized_true_wind_speed_weights', filepath=configuration['
            output_path'] + "plots/weights_for_fit_3d_tws.png")
        __plot_histogram(weights_twa_scaled, 'Histogram_of_normalized_true_wind_angle_weights', filepath=configuration['
            output_path'] + "plots/weights_for_fit_3d_twa.png")

    weights_bsp_scaled_and_limited = __limit(weights_bsp_scaled, lower_limit=0.0, upper_limit=5.0)
    weights_tws_scaled_and_limited = __limit(weights_tws_scaled, lower_limit=0.0, upper_limit=5.0)
    weights_twa_scaled_and_limited = __limit(weights_twa_scaled, lower_limit=0.0, upper_limit=5.0)

    if configuration['plot_weight_histograms'] is True:
        __plot_histogram(weights_bsp_scaled_and_limited, 'Histogram_of_normalized_boat_speed_weights', filepath=configuration['
            output_path'] + "plots/weights_for_fit_3d_bsp.png")
        __plot_histogram(weights_tws_scaled_and_limited, 'Histogram_of_normalized_true_wind_speed_weights', filepath=
            configuration['output_path'] + "plots/weights_for_fit_3d_tws.png")
        __plot_histogram(weights_twa_scaled_and_limited, 'Histogram_of_normalized_true_wind_angle_weights', filepath=
            configuration['output_path'] + "plots/weights_for_fit_3d_twa.png")

    return weights_bsp_scaled_and_limited, weights_tws_scaled_and_limited, weights_twa_scaled_and_limited


def __invert_and_limit(weights, lower_limit, upper_limit):
    for index, weight in enumerate(weights):
        if weight <= lower_limit or weight > upper_limit or weight == 0.0:
            weights[index] = 0.0
        else:
            weights[index] = 1 / (weight**2)


def __calculate_binary_weight(weights, lower_limit, upper_limit):
    for index, weight in enumerate(weights):
        if weight <= lower_limit or weight > upper_limit or weight == 0.0:
            weights[index] = 0.0
        else:
            weights[index] = 1.0


def __limit(weights, lower_limit, upper_limit):
    weights = weights.copy()
    for index, weight in enumerate(weights):
        if weight < lower_limit:
            weights[index] = lower_limit
        elif weight > upper_limit:
            weights[index] = upper_limit
        else:
            weights[index] = weight
    return weights


def __plot_histogram(x, title, filepath=None, show_plot=False):
    plt.figure(figsize=(20, 10))
    n, bins, patches = plt.hist(x, 50, facecolor='green', alpha=0.75)
    plt.title(title)
    plt.grid(True)
    plt.draw()

    if filepath is not None:
        plt.savefig(filepath)
    if show_plot:
        plt.show()


def __calculate_standard_deviations(X_train, y_train):
    if len(X_train) != len(y_train):
        raise ValueError("Arrays␣should␣have␣equal␣length")

    weights_bsp = []
    weights_tws = []
    weights_twa = []

    window_size = 13
    for index, entry in enumerate(X_train):
        if index + 2 >= window_size and index + 2 < len(X_train):
            X_window = X_train[index + 2 - window_size:index + 2]
            bsp_window = y_train[index + 2 - window_size:index + 2]
            tws_window = X_window[:, 0]
            twa_window = X_window[:, 1]

            bsp_weight = __calculate_std(bsp_window)
            tws_weight = __calculate_std(tws_window)
            twa_weight = __calculate_std(twa_window)

        else:
            bsp_weight = 0.0
            tws_weight = 0.0
            twa_weight = 0.0

        weights_bsp.append(bsp_weight)
        weights_tws.append(tws_weight)
        weights_twa.append(twa_weight)

    return weights_bsp, weights_tws, weights_twa


def __calculate_std(array):
    std = numpy.std(array)
    return std
```

Listing A.21: data˙analysis/polar˙diagram˙plotter.py

```
import numpy
import pandas
from matplotlib import rc_context, colors
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from scipy import mean

from polar_diagram_generator.converter import wind_converter
from polar_diagram_generator.util.strings import TWA_TICK_LABELS_WITH_NAME

RC_DEFINITION = {}

LINE_COLORS = [colors.rgb2hex([0.0, 0.0, 0.5]),
               colors.rgb2hex([0.0, 0.0, 1]),
               colors.rgb2hex([0.0, 0.38, 1]),
               colors.rgb2hex([0.0, 0.83, 1]),
               colors.rgb2hex([0.3, 1.0, 0.67]),
               colors.rgb2hex([1.0, 0.48, 0]),
               colors.rgb2hex([1.0, 0.07, 0]),
               colors.rgb2hex([0.5, 0.0, 0])
               ]


class PolarFit(object):
    other_angles = [52, 60, 70, 75, 80, 90, 110, 120, 135, 150, 165, 180]

    velocities = [(6, 44.2, [4.61, 5.1, 5.42, 5.64, 5.69, 5.71, 5.62, 4.96, 4.42, 3.65, 3.25, 3.04, 2.95]),
                  (8, 42.7, [5.45, 6.09, 6.43, 6.67, 6.72, 6.73, 6.64, 5.93, 5.43, 4.70, 4.26, 4.01, 3.91]),
                  (10, 42.3, [6.17, 6.91, 7.22, 7.40, 7.43, 7.44, 7.37, 6.74, 6.25, 5.57, 5.16, 4.90, 4.79]),
                  (12, 42.1, [6.79, 7.45, 7.67, 7.79, 7.81, 7.82, 7.76, 7.36, 6.97, 6.33, 5.94, 5.69, 5.57]),
                  (14, 40.7, [6.98, 7.69, 7.91, 8.05, 8.08, 8.08, 8.03, 7.73, 7.48, 7.01, 6.65, 6.40, 6.28]),
                  (16, 39.6, [7.06, 7.79, 8.01, 8.23, 8.29, 8.32, 8.28, 7.99, 7.81, 7.51, 7.26, 7.05, 6.94]),
                  (20, 39.6, [7.20, 7.87, 8.1, 8.38, 8.52, 8.66, 8.78, 8.49, 8.31, 8.11, 7.98, 7.88, 7.82])]

    def __init__(self, vpp, estimator, start_end_tuples):
        self.start_end_tuples = start_end_tuples
        self.wind_speeds = [6., 8., 10., 12., 14., 16., 20.]
        self.model = estimator
        if vpp is not None:
            self.velocities = vpp
        self.data = pandas.DataFrame()
        possible_angles = sorted([39.0])
        self.data[('index', 0)] = pandas.Series(numpy.ones(len(possible_angles) + len(self.other_angles)), index=numpy.append(
             possible_angles, self.other_angles, axis=0))

        for this_tws, best_twa, bsp in self.velocities:
            vpp_series = pandas.Series(bsp, index=numpy.append([best_twa], self.other_angles, axis=0))
            fit_series = pandas.Series(index=numpy.append([best_twa], self.other_angles, axis=0))
            for this_twa in self.__x_data_for_spline_interpolation(this_tws):
                if self.start_of_fitted_spline(this_tws) <= this_twa <= self.end_of_fitted_spline(this_tws):
                    fit_series[this_twa] = self.model.predict([this_tws, this_twa])[0]
            self.data[('fit', this_tws)] = fit_series
            self.data[('vpp', this_tws)] = vpp_series
        print(self.data)

    def __x_data_for_spline_interpolation(self, tws):
        for this_tws, best_twa, bsp in self.velocities:
            if this_tws == tws:
                start_for_spline = self.start_of_fitted_spline(tws)
                if start_for_spline < best_twa:
                    return numpy.append([start_for_spline, best_twa], self.other_angles, axis=0)
                else:
                    return numpy.append([best_twa], self.other_angles, axis=0)
        return []

    def x_data_for_spline_interpolation(self, tws):
        return self.data['fit',tws].index

    def y_data_for_spline_interpolation(self, tws):
        bsp_array = []
        percent = min(1.0, mean(1.0 / self.data['vpp', tws] * self.data['fit', tws]))

        for key in self.data['fit', tws].index:
            fitted_value = self.data['fit', tws][key]
            vpp_value = self.data['vpp', tws][key]
            if fitted_value is not None and not numpy.math.isnan(fitted_value):
                if key + 15 >= self.end_of_fitted_spline(tws):
                    bsp_array.append(fitted_value - 0.33 * (fitted_value - vpp_value * percent))
                elif key + 5 >= self.end_of_fitted_spline(tws):
                    bsp_array.append(fitted_value - 0.66 * (fitted_value - vpp_value * percent))
                else:
                    bsp_array.append(fitted_value)
            elif vpp_value is not None and not numpy.math.isnan(vpp_value):
                bsp_array.append(vpp_value * percent)
            elif len(bsp_array) > 0:
                bsp_array.append(bsp_array[-1])
            else:
                bsp_array.append(0.0)
        return bsp_array

    def __minimal_bsp(self, tws, twa_list, bsp_orc):
        min_bsp = []
        for index, twa in enumerate(twa_list):
            if self.start_of_fitted_spline(tws) <= twa <= self.end_of_fitted_spline(tws):
                min_bsp.append(min(self.model.predict([tws, twa]), bsp_orc[index]))
            else:
                min_bsp.append(bsp_orc[index])
        return min_bsp

    def start_of_fitted_spline(self, tws):
        for this_tws, start, end in self.start_end_tuples:
            if this_tws == tws:
                return max(start, 39.0)
        return 0

    def end_of_fitted_spline(self, tws):
        for this_tws, start, end in self.start_end_tuples:
            if this_tws == tws:
                return end
        return 180


def plot_polar_fit(data: PolarFit, configuration, name="fit"):
    with rc_context(RC_DEFINITION):
        fig = plt.figure(figsize=(15, 10))
        ax = plt.subplot(111, projection='polar')
        title = 'Polar_Diagram_-_%s' % name
        ax.set_title('%s' % title, va='bottom')
        ax.set_theta_direction(-1)
        ax.set_theta_offset(numpy.pi / 2.0)
        ax.set_xticklabels(TWA_TICK_LABELS_WITH_NAME)  # Twa
        ax.set_yticklabels([2, 4, 6, 8, '10_Boat_Speed_[kn]'])  # Bsp
        ax.set_ylim(0, 10)

        for index, speed in enumerate(data.wind_speeds):

            spline_x = data.x_data_for_spline_interpolation(speed)
            spline_x_polar = wind_converter.wind_angle_cart_2_pol(spline_x)
            spline_y = data.y_data_for_spline_interpolation(speed)
            interpolated_function = interp1d(spline_x_polar, spline_y, kind='cubic')

            pol1 = wind_converter.wind_angle_cart_2_pol(data.start_of_fitted_spline(speed))
            pol2 = wind_converter.wind_angle_cart_2_pol(data.end_of_fitted_spline(speed))

            if min(spline_x_polar) < pol1:
                spline_part1 = numpy.linspace(min(spline_x_polar), pol1, num=41, endpoint=True)
                plt.plot(spline_part1, interpolated_function(spline_part1), '--', color=LINE_COLORS[index], alpha=0.9)

            spline_part2 = numpy.linspace(pol1, pol2, num=41, endpoint=True)
            plt.plot(spline_part2, interpolated_function(spline_part2), '-', color=LINE_COLORS[index], alpha=1.0, label="%d_kn"
                  % speed)

            if pol2 < max(spline_x_polar):
                spline_part3 = numpy.linspace(pol2, max(spline_x_polar), num=41, endpoint=True)
                plt.plot(spline_part3, interpolated_function(spline_part3), '--', color=LINE_COLORS[index], alpha=0.9)

        ax.axis([0, 2 * numpy.pi, 0, 11])

        plt.legend(loc=2, frameon=False, bbox_to_anchor=(1.05, 1),
                   title='True_Wind_Speed:')
        plt.draw()
        plt.savefig(configuration['output_path'] + 'plots/fit_3d_polar_pretty_' + name + '.png')
        if configuration['show_plots'] is True:
            plt.show()
    return len(data.wind_speeds)
```

Listing A.22: data˙analysis/estimator/nonlinear˙estimator.py

```
"""
Custom estimator for non-linear regression.
Allows fits with 2 dimensions.
Does not support errors in multiple variables.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging

import numpy as np
from scipy.optimize import curve_fit
from sklearn.base import BaseEstimator
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted

logger = logging.getLogger('polar.data_analysis.estimator')


class NonlinearEstimator(BaseEstimator):

    popt = None
    pcov = None

    def __init__(self, objective_func):
        self.objective_func = objective_func
        self.X_ = None
        self.y_ = None

    def fit(self, X, y):
        X, y = check_X_y(X, y)
        X = np.ravel(X).T
        y = np.ravel(y).T

        self.popt, self.pcov = curve_fit(self.objective_func, X, y, method='lm', maxfev=800000)
        logger.debug("Nonlinear_Fit_Result:_")
        logger.debug(self.popt)

        self.__calculate_and_print_chi_squared(X, y, self.objective_func, self.popt, 0.3)

        self.X_ = X
        self.y_ = y
        return self

    def predict(self, X):
        check_is_fitted(self, ['X_', 'y_'])
        X = check_array(X)

        X = np.ravel(X).T

        func_argument_count = self.objective_func.__code__.co_argcount
        if func_argument_count == 1:
            return self.objective_func(X)
        elif func_argument_count == 2:
            return self.objective_func(X, self.popt[0])
        elif func_argument_count == 3:
            return self.objective_func(X, self.popt[0], self.popt[1])
        elif func_argument_count == 4:
            return self.objective_func(X, self.popt[0], self.popt[1], self.popt[2])
        elif func_argument_count == 5:
            return self.objective_func(X, self.popt[0], self.popt[1], self.popt[2], self.popt[3])
        elif func_argument_count == 6:
            return self.objective_func(X, self.popt[0], self.popt[1], self.popt[2], self.popt[3], self.popt[4])
        elif func_argument_count == 7:
            return self.objective_func(X, self.popt[0], self.popt[1], self.popt[2], self.popt[3], self.popt[4], self.popt[5])
        elif func_argument_count == 8:
            return self.objective_func(X, self.popt[0], self.popt[1], self.popt[2], self.popt[3], self.popt[4], self.popt[5],
                    self.popt[6])
        else:
            raise TypeError("Only functions up to 6 arguments are allowed")

    def __calculate_and_print_chi_squared(self, xdata, ydata, func, popt, xerror):
        chi_squared = np.sum(((func(xdata, *popt) - ydata) / xerror) ** 2)
        reduced_chi_squared = (chi_squared) / (len(xdata) - len(popt))
        logger.debug('The degrees of freedom for this test is', len(xdata) - len(popt))
        logger.debug('The chi squared value is: ', ("%.2f" % chi_squared))
        logger.debug('The reduced chi squared value is: ', ("%.2f" % reduced_chi_squared))
```

Listing A.23: data_analysis/estimator/odr_estimator_2d.py

```
"""
Custom estimator for non-linear regression.
Allows fits with 2 dimensions and supports errors in multiple variables.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging

import numpy as numpy
import scipy.odr.odrpack as odrpack
from sklearn.base import BaseEstimator
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted

logger = logging.getLogger('polar.data_analysis.estimator')


class ODREstimator2d(BaseEstimator):
    popt = None
    pcov = None
    output = None
    beta_cor = None
    sum_square = None
    sum_square_delta = None
    sum_square_eps = None
    inv_condnum = None
    rel_error = None

    def __init__(self, objective_func, initial_values, standard_deviations_x, standard_deviations_y,
                 max_iterations=1000):
        self.max_iterations = max_iterations
        self.objective_func = objective_func
        def helper_function(parameters, x):
            return self.objective_func(x, *parameters)

        self.callable_objective_func = helper_function

        self.model = odrpack.Model(self.callable_objective_func)
        self.initial_values = initial_values
        self.sx = numpy.add(standard_deviations_x, 0.0000000000001)
        self.sy = numpy.add(standard_deviations_y, 0.0000000000001)

    def fit(self, X, y):
        X, y = check_X_y(X, y)

        tws = X[:, 0]

        mydata = self.__create_data(tws, y, self.sx, self.sy)
        myodr = odrpack.ODR(mydata, self.model, beta0=self.initial_values, maxit=self.max_iterations)

        myoutput = myodr.run()
        self.popt = myoutput.beta
        self.pcov = myoutput.cov_beta
        self.sum_square = myoutput.sum_square
        self.sum_square_delta = myoutput.sum_square_delta
        self.sum_square_eps = myoutput.sum_square_eps
        self.inv_condnum = myoutput.inv_condnum
        self.rel_error = myoutput.rel_error

        self.output = myoutput

        data_set_length = len(y)
        self.__print_output(myoutput, data_set_length)

        logger.debug("\nCorrelation␣Matrix␣:")
        cov = myoutput.cov_beta
        cor = numpy.copy(cov)
        for i, row in enumerate(cov):
            for j in range(len(myoutput.beta)):
                cor_i_j = cov[i, j] / numpy.sqrt(cov[i, i] * cov[j, j])
                cor[i, j] = cor_i_j
                print("{0:<␣8.3g}".format(cor_i_j),
                      end="")
            print()
        self.beta_cor = cor

        self.X_ = X
        self.y_ = y
        return self

    def predict(self, X):
        check_is_fitted(self, ['X_', 'y_'])
        X = check_array(X)

        tws = X[:, 0]

        return self.model.fcn(self.popt, (tws))

    def __print_output(self, myoutput, data_set_length):
        logger.debug("ODR␣Fit␣Result:␣")
        myoutput.pprint()
        logger.debug("————————————————————————————")
        logger.debug("Sum␣of␣squared␣errors␣␣␣␣␣␣%s", myoutput.sum_square)
        logger.debug("Sum␣of␣squared␣error␣delta␣%s", myoutput.sum_square_delta)
        logger.debug("Sum␣of␣squared␣error␣eps␣␣␣%s", myoutput.sum_square_eps)
        logger.debug("Quasi␣Chi^2␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%s", myoutput.res_var)
        logger.debug("————————————————————————————")
        weight_matrix = numpy.vstack((1 / self.sy, 1 / self.sx))
        rchi2_min_weighted = numpy.sum(myoutput.eps.T * weight_matrix * myoutput.eps)
        dof = data_set_length - len(self.initial_values)
        logger.debug("Chi^2␣min␣self␣␣␣␣␣␣␣␣␣␣␣␣␣%s", numpy.sum(numpy.power(myoutput.eps, 2)))
        logger.debug("Reduced␣Chi^2␣self␣␣␣␣␣␣␣␣␣%s", numpy.sum(numpy.power(myoutput.eps, 2)) / dof)
        logger.debug("Chi^2␣min␣self␣weighted␣␣␣␣%s", rchi2_min_weighted)
        logger.debug("Reduced␣Chi^2␣min␣weighted␣%s", rchi2_min_weighted / dof)
        logger.debug("————————————————————————————")

    def __create_weighted_data(self, twa, tws, y, sx, sy):
        return odrpack.RealData((tws, twa), y, sx=sx, sy=sy)

    def __create_data(self, tws, y, sx, sy):
        return odrpack.Data((tws), y)
```

Listing A.24: data˙analysis/estimator/odr˙estimator.py

```
"""
Custom estimator for non-linear regression.
Allows fits with 3 dimensions and supports errors in multiple variables.

"""

# Author: Stefan Simon
# License: CC BY-NC-ND 4.0
import logging

import numpy as numpy
import scipy.odr.odrpack as odrpack
from sklearn.base import BaseEstimator
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted

logger = logging.getLogger('polar.data_analysis.estimator')


class ODREstimator(BaseEstimator):
    popt = None
    pcov = None
    output = None
    beta_cor = None
    sum_square = None
    sum_square_delta = None
    sum_square_eps = None
    inv_condnum = None
    rel_error = None

    def __init__(self, objective_func, initial_values, weights_X=None, weights_y=None, max_iterations=1000):
        self.objective_func = objective_func
        self.model = odrpack.Model(objective_func)
        self.initial_values = initial_values
        self.weights_X = weights_X
        self.weights_y = weights_y
        self.max_iterations = max_iterations
        self.X_ = None
        self.y_ = None
        self.dof = None

    def fit(self, X, y):
        X, y = check_X_y(X, y)

        tws = X[:, 0]
        twa = X[:, 1]

        mydata = odrpack.Data((tws, twa), y, wd=self.weights_X, we=self.weights_y)
        myodr = odrpack.ODR(mydata, self.model, beta0=self.initial_values, maxit=self.max_iterations)
        myodr.set_job(fit_type=2)

        myoutput = myodr.run()
        self.popt = myoutput.beta
        self.pcov = myoutput.cov_beta
        self.sum_square = myoutput.sum_square
        self.sum_square_delta = myoutput.sum_square_delta
        self.sum_square_eps = myoutput.sum_square_eps
        self.inv_condnum = myoutput.inv_condnum
        self.rel_error = myoutput.rel_error

        self.output = myoutput

        data_set_length = len(y)
        self.__print_output(myoutput, data_set_length)

        logger.debug("\nCorrelation Matrix :")
        cov = myoutput.cov_beta
        cor = numpy.copy(cov)
        for i, row in enumerate(cov):
            for j in range(len(myoutput.beta)):
                cor_i_j = cov[i, j] / numpy.sqrt(cov[i, i] * cov[j, j])
                cor[i, j] = cor_i_j
                print("{0:< 8.3g}".format(cor_i_j),
                      end="")
            print()
        self.beta_cor = cor

        self.X_ = X
        self.y_ = y
        return self

    def predict(self, X):
        check_is_fitted(self, ['X_', 'y_'])
        X = check_array(X)

        tws = X[:, 0]
        twa = X[:, 1]

        return self.objective_func(self.popt, (tws, twa))

    def __print_output(self, myoutput, data_set_length):
        logger.debug("ODR Fit Result: ")
        myoutput.pprint()
        logger.debug("-----------------------------------------")
        logger.debug("Sum of squared errors (chi^2) %s", myoutput.sum_square)
        logger.debug("Sum of squared error delta    %s", myoutput.sum_square_delta)
        logger.debug("Sum of squared error eps      %s", myoutput.sum_square_eps)
        logger.debug("reduced chi^2                 %s", myoutput.res_var)
        logger.debug("-----------------------------------------")
        self.dof = data_set_length - len(self.initial_values)
        logger.debug("#data                         %s", data_set_length)
        logger.debug("variables                     %s", len(self.initial_values))
        logger.debug("DOF                           %s", self.dof)
        logger.debug("Chi^2 min self                %s", numpy.sum(numpy.power(myoutput.eps, 2)))
        logger.debug("Reduced Chi^2 self            %s", numpy.sum(numpy.power(myoutput.eps, 2)) / self.dof)
        logger.debug("-----------------------------------------")
        logger.debug("Chi^2 min self NEW            %s",
               numpy.sum(numpy.power(myoutput.eps, 2)) + numpy.sum(numpy.power(myoutput.delta, 2)))
        logger.debug("Reduced Chi^2 self NEW        %s",
               (numpy.sum(numpy.power(myoutput.eps, 2)) + numpy.sum(numpy.power(myoutput.delta, 2))) / self.dof)
        logger.debug("-----------------------------------------")
```

Listing A.25: data˙analysis/models/models2d.py

```
import numpy


def all_combination_models():
    return [
        ("no combination", None),
        ("linear combination", polynomial_1_degree)
    ]


def all_models():
    return [
        ("2 degree polynomial", polynomial_2_degrees),
        ("3 degree polynomial", polynomial_3_degrees),
        ("4 degree polynomial", polynomial_4_degrees),
        ("6 degree polynomial", polynomial_6_degrees),
        ("Inverted parabola", inverted_parabola),
        ("Concave with downturn", concave_with_downturn),
        ("Concave with saturation limit", concave_with_saturation_limit),
        ("Concave with saturation limit and downturn", concave_with_saturation_limit_and_downturn),
        ("S-shaped with saturation limit", s_shaped_with_saturation_limit),
        ("S-shaped with saturation limit and downturn", s_shaped_with_saturation_limit_and_downturn),
        ("Gompertz model", gompertz_model),
        ("Gaussian model", gaussian_model),
        ("Gaussian model with offset", gaussian_model_with_offset),
        ("Gaussian mixture model", gmm_model),
    ]


def polynomial_1_degree(x, a, b):
    return a + b * x


def polynomial_2_degrees(x, a, b, c):
    return a + b * x + c * pow(x, 2)


def polynomial_3_degrees(x, a, b, c, d):
    return a + b * x + c * pow(x, 2) + d * pow(x, 3)


def polynomial_4_degrees(x, a, b, c, d, e):
    return a + b * x + c * pow(x, 2) + d * pow(x, 3) + e * pow(x, 4)
```

```
def polynomial_6_degrees(x, a, b, c, d, e, f, g):
    return a + b * x + c * pow(x, 2) + d * pow(x, 3) + e * pow(x, 4) + f * pow(x, 5) + g * pow(x, 6)


def inverted_parabola(x, a, b, c): return b + c * pow(x - a, 2)


def concave_with_downturn(x, a, b, c): return a + b * x - c * pow(x, 2)


def concave_with_saturation_limit(x, a, b, c, d): return c - a * numpy.exp(-b * x)


def concave_with_saturation_limit_and_downturn(x, a, b, c, d): return c - a * numpy.exp(-b * x) - d * pow(x, 2)


def s_shaped_with_saturation_limit(x, a, b, c, d): return c / (1 + numpy.exp(a - b * x))


def s_shaped_with_saturation_limit_and_downturn(x, a, b, c, d): return c / (1 + numpy.exp(a - b * x)) - d * pow(x, 2)


def gompertz_model(x, a, b, c): return c * numpy.exp(- a * pow(b, x))  # nach BACKHAUS

# def gompertz_model(x, a, b, c): return c * numpy.exp(- a * numpy.exp(-b * x))  # nach Wikipedia


def gaussian_model(x, a, b, c): return a * numpy.exp(-0.5 * pow((x-b), 2) / c)


def gaussian_model_with_offset(x, a, b, c, z): return a * numpy.exp(-0.5 * pow((x-b), 2) / c) + z


def gmm_model(x, a, b, c, d, e, f, z): return a * numpy.exp(-0.5 * pow((x-b), 2) / (c)) + d * numpy.exp(-0.5 * pow((x-e), 2) /
    f) + z
```

Listing A.26: data˙analysis/models/models3d.py

```
import numpy

from numpy.ma import exp
from scipy.odr import odrpack


class DynamicModel(odrpack.Model):

    def __init__(self, model_tws, model_twa, model_tws_times_twa=None):
        self.model_tws = model_tws
        self.model_twa = model_twa
        self.model_tws_times_twa = model_tws_times_twa

        def helper_function(parameters, x):
            return 0

        super().__init__(helper_function)


class DynamicCallableModel:

    def __init__(self, model_tws, model_twa, model_tws_times_twa=None):
        self.model_tws = model_tws
        model_tws_function_length = self.model_tws.__code__.co_argcount-1
        self.model_twa = model_twa
        model_twa_function_length = self.model_twa.__code__.co_argcount-1
        self.model_tws_times_twa = model_tws_times_twa
        if self.model_tws_times_twa is not None:
            model_tws_times_twa_function_length = self.model_tws_times_twa.__code__.co_argcount-1
        else:
            model_tws_times_twa_function_length = 0
        self.first_end = model_tws_function_length
        self.second_end = self.first_end + model_twa_function_length
        self.third_end = self.second_end + model_tws_times_twa_function_length

    def __call__(self, parameters, x):
        try:
            tws, twa = x
        except:
            tws = x[:, 0]
            twa = x[:, 1]
        tws_value = self.model_tws(tws, *parameters[0:self.first_end])
        twa_value = self.model_twa(twa, *parameters[self.first_end:self.second_end])
        if self.model_tws_times_twa is not None:
            tws_times_twa_value = self.model_tws_times_twa(tws * twa, *parameters[self.second_end:self.third_end])
        else:
            tws_times_twa_value = 0
        return tws_value + twa_value + tws_times_twa_value

    def initial_parameters(self):
        return numpy.ones(self.third_end).tolist()
        # return numpy.ndarray((self.third_end,),float)+1


def model_tws_concave_with_downturn_and_twa(parameters, x):
    a, b, c, d, e, f, g = parameters
    try:
        tws, twa = x
    except:
        tws = x[:, 0]
        twa = x[:, 1]
    return a + b * tws - c * pow(tws, 2) + d * twa - pow(twa - e, 2) * f + g * tws * twa


def derivatives_model_tws_concave_with_downturn_and_twa(parameters, data, dflags):
    x, y, err, weights_bsp, weights_tws, weights_twa = data           # Data arrays is a tuple given by programmer
    tws = x[:, 0]
    twa = x[:, 1]
    a, b, c, d, e, f, g = parameters               # Parameters which are adjusted by kmpfit
    pderiv = numpy.zeros([len(parameters), len(x)])
    for i, flag in enumerate(dflags):
        if flag:
            if i == 0:  # a
                pderiv[0] = 1.0  # copy from derive_model
            elif i == 1:  # b
                pderiv[1] = tws
            elif i == 2:  # c
                pderiv[2] = - pow(tws, 2)
            elif i == 3:  # d
                pderiv[3] = twa
            elif i == 4:  # e
                pderiv[4] = 2 * f * (- e * twa)
            elif i == 5:  # f
                pderiv[5] = - pow(-e + twa, 2)
            elif i == 6:  # g
                pderiv[6] = tws * twa
    return pderiv/-err

##############################################################


def model_tws_times_twa_s_shaped_with_downturn(parameters, x):
    a, b, c, d, m, n, o, p, t = parameters
    try:
        tws, twa = x
    except:
        tws = x[:, 0]
        twa = x[:, 1]
    return c / (1 + exp(a - b * tws)) - d * tws**2 + m / (1 + exp(n - o * twa)) - p * twa**2 + t * tws * twa


def derivatives_model_tws_times_twa_s_shaped_with_downturn(parameters, data, dflags):
    x, y, err, weights_bsp, weights_tws, weights_twa = data           # Data arrays is a tuple given by programmer
    tws = x[:, 0]
    twa = x[:, 1]
    a, b, c, d, m, n, o, p, t = parameters               # Parameters which are adjusted by kmpfit
    pderiv = numpy.zeros([len(parameters), len(x)])
    for i, flag in enumerate(dflags):
        if flag:
            if i == 0:  # a
                pderiv[0] = -c*exp(a - b*tws)/(exp(a - b*tws) + 1)**2  # copy from derive_model
            elif i == 1:  # b
                pderiv[1] = c*tws*exp(a - b*tws)/(exp(a - b*tws) + 1)**2
            elif i == 2:  # c
                pderiv[2] = 1/(exp(a - b*tws) + 1)
            elif i == 3:  # d
                pderiv[3] = -tws**2
            elif i == 4:  # m
```

```
                pderiv[4] = 1/(exp(n - o*twa) + 1)
            elif i == 5:  # n
                pderiv[5] = -m*exp(n - o*twa)/(exp(n - o*twa) + 1)**2
            elif i == 6:  # o
                pderiv[6] = m*twa*exp(n - o*twa)/(exp(n - o*twa) + 1)**2
            elif i == 7:  # p
                pderiv[7] = -twa**2
            elif i == 8:  # t
                pderiv[8] = tws * twa
    return pderiv/-err

##############################################################


def model_tws_s_shaped_with_downturn_and_twa_gaussian(parameters, x):
    a, b, c, d, m, n, o = parameters
    try:
        tws, twa = x
    except:
        tws = x[:, 0]
        twa = x[:, 1]
    return c / (1 + exp(a - b * tws)) - d * tws**2 + m * exp(-0.5 * pow((twa-n) / o, 2))


def derivatives_model_tws_s_shaped_with_downturn_and_twa_gaussian(parameters, data, dflags):
    x, y, err, weights_bsp, weights_tws, weights_twa = data           # Data arrays is a tuple given by programmer
    tws = x[:, 0]
    twa = x[:, 1]
    a, b, c, d, m, n, o = parameters               # Parameters which are adjusted by kmpfit
    pderiv = numpy.zeros([len(parameters), len(x)])
    for i, flag in enumerate(dflags):
        if flag:
            if i == 0:  # a
                pderiv[0] = -c*exp(a - b*tws)/(exp(a - b*tws) + 1)**2  # copy from derive_model
            elif i == 1:  # b
                pderiv[1] = c*tws*exp(a - b*tws)/(exp(a - b*tws) + 1)**2
            elif i == 2:  # c
                pderiv[2] = 1/(exp(a - b*tws) + 1)
            elif i == 3:  # d
                pderiv[3] = -tws**2
            elif i == 4:  # m
                pderiv[4] = exp(-0.5*(-n + twa)**2/o**2)
            elif i == 5:  # n
                pderiv[5] = -0.5*m*(2*n - 2*twa)*exp(-0.5*(-n + twa)**2/o**2)/o**2
            elif i == 6:  # o
                pderiv[6] = 1.0*m*(-n + twa)**2*exp(-0.5*(-n + twa)**2/o**2)/o**3
    return pderiv/-err

##############################################################


def model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination(parameters, x):
    a, b, c, d, m, n, o, t = parameters
    try:
        tws, twa = x
    except:
        tws = x[:, 0]
        twa = x[:, 1]
    return c / (1 + exp(a - b * tws)) - d * tws**2 + m * exp(-0.5 * pow((twa-n) / o, 2)) + t * tws * twa


def derivatives_model_tws_s_shaped_with_downturn_and_twa_gaussian_plus_combination(parameters, data, dflags):
    x, y, err, weights_bsp, weights_tws, weights_twa = data           # Data arrays is a tuple given by programmer
    tws = x[:, 0]
    twa = x[:, 1]
    a, b, c, d, m, n, o, t = parameters               # Parameters which are adjusted by kmpfit
    pderiv = numpy.zeros([len(parameters), len(x)])
    for i, flag in enumerate(dflags):
        if flag:
            if i == 0:  # a
                pderiv[0] = -c*exp(a - b*tws)/(exp(a - b*tws) + 1)**2  # copy from derive_model
            elif i == 1:  # b
                pderiv[1] = c*tws*exp(a - b*tws)/(exp(a - b*tws) + 1)**2
            elif i == 2:  # c
                pderiv[2] = 1/(exp(a - b*tws) + 1)
            elif i == 3:  # d
                pderiv[3] = -tws**2
            elif i == 4:  # m
                pderiv[4] = exp(-0.5*(-n + twa)**2/o**2)
            elif i == 5:  # n
```

```
                pderiv[5] = -0.5*m*(2*n - 2*twa)*exp(-0.5*(-n + twa)**2/o**2)/o**2
            elif i == 6:  # o
                pderiv[6] = 1.0*m*(-n + twa)**2*exp(-0.5*(-n + twa)**2/o**2)/o**3
            elif i == 7:  # t
                pderiv[7] = tws * twa
    return pderiv/-err

##############################################################


def model_tws_s_shaped_with_saturation_and_twa_gaussian(parameters, x):
    a, b, c, m, n, o = parameters
    try:
        tws, twa = x
    except:
        tws = x[:, 0]
        twa = x[:, 1]
    return c / (1 + exp(a - b * tws)) + m * exp(-0.5 * pow((twa-n) / o, 2))


def derivatives_model_tws_s_shaped_with_saturation_and_twa_gaussian(parameters, data, dflags):
    x, y, err, weights_bsp, weights_tws, weights_twa = data           # Data arrays is a tuple given by programmer
    tws = x[:, 0]
    twa = x[:, 1]
    a, b, c, m, n, o = parameters                # Parameters which are adjusted by kmpfit
    pderiv = numpy.zeros([len(parameters), len(x)])
    for i, flag in enumerate(dflags):
        if flag:
            if i == 0:  # a
                pderiv[0] = -c*exp(a - b*tws)/(exp(a - b*tws) + 1)**2  # copy from derive_model
            elif i == 1:  # b
                pderiv[1] = c*tws*exp(a - b*tws)/(exp(a - b*tws) + 1)**2
            elif i == 2:  # c
                pderiv[2] = 1/(exp(a - b*tws) + 1)
            elif i == 3:  # m
                pderiv[3] = exp(-0.5*(-n + twa)**2/o**2)
            elif i == 4:  # n
                pderiv[4] = -0.5*m*(2*n - 2*twa)*exp(-0.5*(-n + twa)**2/o**2)/o**2
            elif i == 5:  # o
                pderiv[5] = 1.0*m*(-n + twa)**2*exp(-0.5*(-n + twa)**2/o**3
    return pderiv/-err

##############################################################
```

Bibliography

Acevedo, Miguel F. (2012). *Data Analysis and Statistics for Geography, Environmental Science, and Engineering*. English. CRC Press. 557 pp. ISBN: 978-1-4398-8501-7 (cit. on pp. 20, 22, 23).

Andrae, R., T. Schulze-Hartung, and P. Melchior (2010). "Dos and don'ts of reduced chi-squared." In: *ArXiv e-prints*. arXiv: 1012.3754 [astro-ph.IM] (cit. on p. 35).

Backhaus, Klaus et al. (2016). *Multivariate Analysemethoden. Eine anwendungsorientierte Einführung*. German. 14th ed. Gabler Verlag. Chap. XII. 647 pp. ISBN: 978-3-662-46076-4. DOI: 10.1007/978-3-662-46076-4 (cit. on pp. 24, 29, 33, 36, 37, 66, 67).

Barron, A. (1997). *Statistics 101-103, Introduction to Statistics - Chi-Square Goodness of Fit Test*. URL: http://www.stat.yale.edu/Courses/1997-98/101/chigf.htm (visited on 08/25/2017) (cit. on p. 35).

Bavaria Cruiser 40S (2016). URL: http://www.roundpalagruza.at/20x-bavaria-cruiser-40s/ (visited on 03/11/2017) (cit. on p. 61).

Behnel, S. et al. (2011). "Cython: The Best of Both Worlds." In: *Computing in Science Engineering* 13.2, pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118 (cit. on p. 54).

Betke, Klaus (2001). *The NMEA 0183 Protocol*. URL: http://www.tronico.fi/OH6NT/docs/NMEA0183.pdf (visited on 08/04/2016) (cit. on pp. 42–44).

Buitinck, Lars et al. (2013). "API design for machine learning software: experiences from the scikit-learn project." In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122 (cit. on p. 55).

Cython License (2009). URL: https://github.com/cython/cython/blob/master/LICENSE.txt (visited on 11/17/2017) (cit. on p. 54).

Department of Statistics Online Programs (2017). *Polynomial Regression*. Department of Statistics Online Programs, The Pennsylvania State University. URL: https://onlinecourses.science.psu.edu/stat501/node/324 (visited on 07/10/2017) (cit. on p. 37).

Die Route des Round Palagruža Cannonballs (2010). URL: http://www.roundpalagruza.at/die-route-des-rpc/ (visited on 07/28/2017) (cit. on p. 60).

Expedition 10 (2017). URL: http://www.expeditionmarine.com/ (visited on 07/14/2017) (cit. on p. 44).

Fernandes, Richard and Sylvain G. Leblanc (2005). "Parametric (modified least squares) and non-parametric (Theil–Sen) linear regressions for predicting biophysical parameters in the presence of measurement errors." In: *Remote Sensing of Environment* 95.3, pp. 303–316. ISSN: 0034-4257. DOI: http://dx.doi.org/10.1016/j.rse.2005.01.005. URL: http://www.sciencedirect.com/science/article/pii/S0034425705000404 (cit. on pp. 25, 26).

Frost, Jim (2011). *Linear or Nonlinear Regression? That Is the Question.* URL: http://blog.minitab.com/blog/adventures-in-statistics-2/linear-or-nonlinear-regression-that-is-the-question (visited on 05/11/2017) (cit. on pp. 28, 29).

Gamma, Erich et al. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2 (cit. on p. 54).

Garmin (2017). *Garmin - What is GPS?* URL: http://www8.garmin.com/aboutGPS/ (visited on 09/19/2017) (cit. on p. 42).

Garrett, R. (1996). *The Symmetry of Sailing: The Physics of Sailing for Yachtsmen*. Sheridan House. ISBN: 9781574090000. URL: https://books.google.at/books?id=0VLXORumEF4C (cit. on pp. 9, 11, 12, 16).

General Information (2017). URL: https://docs.python.org/3/faq/general.html (visited on 04/26/2017) (cit. on p. 47).

Guerzhoy, Michael (2015). *Training, test, and validation sets. CSC320: Introduction to Visual Computing*. URL: http://www.cs.toronto.edu/~guerzhoy/320/lec/training.pdf (visited on 08/26/2017) (cit. on p. 38).

Gutschi, Clemens (2015). "Course Optimization for Sailing Yachts. Weather Routing and Real Time Performance Visualization." master's thesis. Institute of Engineering and Business Informatics (cit. on p. 94).

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning. Data Mining, Inference, and Prediction, Second Edition*. 2nd ed. Springer Series in Statistics. Springer New York Inc. 745 pp. ISBN: 9780387848587. DOI: 10.1007/978-0-387-84858-7 (cit. on p. 23).

Hodges Marine Electronics (2017). *B&G 608 Wind Sensor*. URL: https://www.hodgesmarine.com/B-G-608-Wind-Sensor-W-O-Cable-p/bag000-13714-001.htm (visited on 07/10/2017) (cit. on p. 41).

Hwaci (2017). *Most Widely Deployed and Used Database Engine*. URL: https://www.sqlite.org/mostdeployed.html (visited on 08/18/2017) (cit. on p. 50).

Inc., Apple (2017). *MacBook Pro Tech Specs*. URL: https://www.apple.com/macbook-pro/specs/ (visited on 10/08/2017) (cit. on p. 47).

International Measurement System IMS 2016 (2016). URL: http://www.orc.org/rules/IMS%202016.pdf (visited on 01/02/2017) (cit. on pp. 1, 13).

Jones, Eric, Travis Oliphant, Pearu Peterson, et al. (2001–). *SciPy: Open source scientific tools for Python*. [Online; accessed ¡today¿]. URL: http://www.scipy.org/ (cit. on p. 54).

Kapteyn Astronomical Institute (2015). *ODR and kmpfit with weighted fit*. URL: https://www.astro.rug.nl/software/kapteyn/kmpfittutorial.html#orthogonal-distance-regression-odr (visited on 07/07/2017) (cit. on p. 34).

Klasing, Delius, ed. (2016). *Seemannschaft. Handbuch für den Yachtsport*. German. 31st ed. Bielefeld. 816 pp. ISBN: 978-3-7688-3248-9 (cit. on pp. 1, 3, 5, 19).

Microsoft Cooperation (2009). *Microsoft Application Architecture Guide (Patterns & Practices)*. Microsoft Press; Second Edition edition. Chap. Chapter 5: Layered Application Guidelines. 560 pp. ISBN: 978-0735627109. URL: https://msdn.microsoft.com/en-us/library/ee658109.aspx (cit. on p. 46).

MPX-Config2 for MiniPlex-2 (2015). URL: http://www.shipmodul.com/en/downloads.html (visited on 03/14/2017) (cit. on pp. 44, 61).

National Institute of Standards and Technology (2013). *Weighted Least Squares Regression*. English. URL: http://www.itl.nist.gov/div898/handbook/pmd/section1/pmd143.htm (visited on 08/18/2017) (cit. on p. 31).

Nealen, Andrew (2004). *An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation*. English. URL: http://www.nealen.com/projects/mls/asapmls.pdf (visited on 02/11/2017) (cit. on p. 27).

NMEA 0183 and Multiplexers (2017). URL: http://www.shipmodul.com/en/multiplexer.html (visited on 03/14/2017) (cit. on p. 45).

Nocedal, Jorge (2006). *Numerical Optimization*. English. Springer-Verlag New York. Chap. XXII. 664 pp. ISBN: 978-0-387-30303-1. DOI: 10.1007/978-0-387-40065-5 (cit. on pp. 27, 30–33).

ORC Club Certificate Bavaria CR 40S (2012). URL: http://www.roundpalagruza.org/wp-content/uploads/cro1845_capivari-120517.pdf (visited on 03/11/2017) (cit. on pp. 14, 15, 18, 60).

ORC International Certificate (2017). URL: http://orc.org/index.asp?id=23 (visited on 10/11/2017) (cit. on p. 1).

ORC VPP Documentation 2016 (2016). URL: http://www.orc.org/rules/ORC%20VPP%20Documentation%202016.pdf (visited on 01/02/2017) (cit. on pp. 1, 13).

Pedregosa, F. et al. (2011). “Scikit-learn: Machine Learning in Python.” In: *Journal of Machine Learning Research* 12, pp. 2825–2830 (cit. on p. 55).

Püschl, Wolfgang (2012). *Physik des Segelns. Wie Segeln wirklich funktioniert*. German. Ed. by Wiley-VCH Verlag GmbH & Co. KGaA. Wiley-VCH Verlag GmbH & Co.

KGaA. 265 pp. ISBN: 9783527411061. DOI: 10.1002/9783527648481 (cit. on pp. 1, 5–7, 10, 11, 17, 18).

Python 2 or Python 3 (2017). URL: https://wiki.python.org/moin/Python2orPython3 (visited on 08/17/2017) (cit. on p. 47).

Round Palagruža Cannonball, Medien-Info 3 (2016). *Drei Klassen – drei Sieger: Fuczik, Lagger, Brückner*. URL: http://www.roundpalagruza.at/wp-content/uploads/2016/04/2006-04-15_MEDIENINFO_No.3_Round-Palagruza-Cannonball.pdf (visited on 07/28/2017) (cit. on p. 64).

S2ware, LLC (2017). *Furuno ST-02MSB Bronze Thru-Hull Speed and Temp Sensor*. URL: http://www.yachtsofstuff.com/yosproduct.asp?ypid=8794&level=46 (visited on 07/10/2017) (cit. on p. 41).

Sarle, Warren S. (2002). *AI: Frequently Asked Questions - Neuronal Networks. What are the population, sample, training set, design set, validation set, and test set?* URL: ftp://ftp.sas.com/pub/neural/FAQ.html#A_data (visited on 08/24/2017) (cit. on p. 38).

Scheer, Elke (2010). "Auftrieb durch Wasser und Wind. Physik des Segelns." In: *Physik in unserer Zeit* 41.4, pp. 184–190. ISSN: 1521-3943. DOI: 10.1002/piuz.201001237. URL: http://dx.doi.org/10.1002/piuz.201001237 (cit. on p. 9).

scikit-learn-developers (2016). *An introduction to machine learning with scikit-learn*. URL: http://scikit-learn.org/stable/tutorial/basic/tutorial.html (visited on 03/22/2017) (cit. on pp. 27, 38).

Shalizi, Cosma (2009). *Extending Linear Regression: Weighted Least Squares, Heteroskedasticity, Local Polynomial Regression*. Statistics Department, Carnegie Mellon University. URL: http://www.stat.cmu.edu/~cshalizi/350/lectures/18/lecture-18.pdf (visited on 05/12/2017) (cit. on p. 32).

ShipModule MiniPlex-2USB (2015). URL: http://www.shipmodul.de/produkte/miniplex-2usb.html (visited on 03/14/2017) (cit. on p. 44).

Stelzer, Roland and Tobias Pröll (2008). "Autonomous sailboat navigation for short course racing." In: *Robotics and Autonomous Systems* 56.7, pp. 604–614. ISSN: 0921-8890. DOI: http://dx.doi.org/10.1016/j.robot.2007.10.004. URL: //www.sciencedirect.com/science/article/pii/S0921889007001480 (cit. on p. 13).

Susmel, Rauli (2014). *Econometrics II: Quantitative Methods in Finance II (FINA 8397) - Lecture 12 - Nonparametric Regression*. Bauer College of Business, University of Houston. URL: http://www.bauer.uh.edu/rsusmel/phd/ec1-27.pdf (cit. on p. 25).