Hausarbeit

DLMDWPMP01 - Programmieren mit Python

Erstellt von:	Steffen Baumann
Matrikelnummer:	IU14089925
Studiengang:	Informatik (Master of Science)
Tutor:	Dr. Thomas Kopsch

vorgelegt am:	16.04.2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Programmiersprache Python ist eine einfach zu erlernende Sprache, die es dem Anwender erlaubt, komplexe Programme für verschiedene Anwendungsgebiete zu entwickeln (Theis, 2017, S.17). Die Grundlagen dieser Hausarbeit bilden drei CSV-Dateien, die vier Trainingsfunktionen („train.csv“), 50 Idealfunktionen („ideal.csv“) sowie 100 Testpunkte („test.csv“) beinhalten. Ziel dieser Hausarbeit ist es, ein Python-Programm zu entwickeln, das die drei CSV-Dateien importiert, den Inhalt verarbeitet und abschließend die Daten in eine SQLite-Datenbank speichert.

Die folgenden zwei Kapitel geben die Aufgabenstellung, unterteilt in zwei Aufgabenteile, wieder. Im Hauptteil werden die verwendeten Bibliotheken vorgestellt sowie die Struktur des Programms erklärt. Im Schlussteil wird ein Resümee gezogen, wobei auch darauf eingegangen wird, welche Vor- sowie Nachteile das erzeugte Python-Programm besitzt.

1.1 Aufgabenteil 1

In einem ersten Abschnitt soll zu den vier Trainingsfunktionen jeweils eine bestmögliche Idealfunktion ermittelt werden. Eine Idealfunktion ist dann die bestmögliche Passung, wenn die Summe aller quadratischen y-Abweichungen minimal ist. Dieses Kriterium kann mit folgender Formel (Least-Square) berechnet werden:

$$y_A = \frac{1}{N}\sum_{i}^{N}(y_i - \bar{y_i})^2 \tag{1.1}$$

y_i = Y-Wert der Trainingsfunktion
$\bar{y_i}$ = Y-Wert der Idealfunktion

Sind die besten Passungen aus Trainings- und Idealfunktion bestimmt, sollen die jeweiligen Ergebnisse logisch visualisiert werden.

1.2 Aufgabenteil 2

Die vier bestmöglichen Idealfunktionen aus Aufgabenteil 1 werden nun zur Analyse der 100 Testpunkte verwendet. Es wird geprüft, ob die Testpunkte einer oder mehreren Idealfunktionen zugeordnet werden können. Um zu entscheiden, ob ein Testpunkt zu einer Idealfunktion passt, wird zuerst die Abweichung zwischen dem Y-Wert des Testpunktes und dem Y-Wert der Idealfunktion berechnet. Ist die Abweichung kleiner als das Testkriterium (maximale Y-Abweichung zwischen Ideal – und Trainingsfunktion, multipliziert mit $\sqrt{2}$), gilt der Testpunkt als anpassbar. Auch die Ergebnisse aus Aufgabenteil 2 sollen logisch visualisiert werden.

2 Python-Bibliotheken und -Module

Dieses Kapitel beschreibt die in dieser Hausarbeit verwendeten Python-Bibliotheken sowie -Module und erläutert, welche Abläufe im Programm-Code sie vereinfachen.

2.1 Pandas-Bibliothek

Die Pandas-Bibliothek ist spezialisiert für das umfangreiche Arbeiten mit strukturierten oder tabellarischen Daten. Sie eignet sich besonders gut für die Manipulation, Vorbereitung und Bereinigung von Daten (McKinney, 2023, S.23).

Für die Bearbeitung der vorliegenden Aufgabenstellung wird das Pandas-Dataframe inklusive der anwendbaren Funktionen verwendet. Ein Pandas-Dataframe ist eine tabellenartige, geordnete Datenstruktur, wobei jede Zeile sowie Spalte einen Index besitzt (McKinney, 2023, S.148).

In dieser Hausarbeit wird die Pandas-Bibliothek verwendet, um die Daten aus den CSV-Dateien zu importieren und zu speichern. Mit Hilfe der zur Pandas-Bibliothek gehörenden Methoden werden die erzeugten Pandas-Dataframes bearbeitet und so manipuliert, dass sie die Weiterverarbeitung zum Erreichen des Ziels der Aufgabenstellung ermöglichen.

2.2 Matplotlib-Bibliothek

Matplotlib ist eine der populärsten Bibliotheken zur Visualisierung von Daten in Python. Sie ermöglicht es dem Anwender, verschiedene Arten von Diagrammen und Grafiken zu erstellen. Hierzu zählen unter anderem Scatterplots, Liniendiagramme und Balkendiagramme (Matthes, 2019, S.306). Die Matplotlib-Bibliothek besitzt die Fähigkeit, mit einer Vielzahl von Betriebssystemen und grafischen Ausgabegeräten zu interagieren (VanderPlas, 2018, S.245).

Alle in dieser Hausarbeit erstellten Grafiken zur Visualisierung der Ergebnisse werden auf Basis der Matplotlib-Bibliothek erstellt. Für Aufgabenteil 1 werden ausschließlich Liniendiagramme erstellt. Die Ergebnisse aus Aufgabenteil 2 werden in einer Grafik visualisiert, die ein Liniendiagramm und einen Scatterplot kombiniert.

Für eine bessere Lesbarkeit der Diagramme/Grafiken werden Titel, Legende und Achsenbeschriftung hinzugefügt (Matthes, 2019, S.308).

2.3 SQL-Alchemy-Bibliothek

Die SQL-Alchemy-Bibliothek dient als Verbindungselement zwischen dem Python-Code und einer Datenbank. SQL-Alchemy konvertiert Python-Funktionsaufrufe in die entsprechenden SQL-Anweisungen. Es werden mehrere Datenbanken, bspw. MySQL oder SQLite, unterstützt (Campesato, 2023, S.253).

In dieser Hausarbeit sollen die Ergebnisse in einer SQLite-Datenbank gespeichert werden. SQLite ist eine prozessinterne Bibliothek, die eine in sich geschlossene, serverlose, konfigurationsfreie SQL-Datenbank-Engine beinhaltet. Der Code für SQLite ist öffentlich zugänglich, wobei die SQLite-Datenbank die am weitesten verbreitete Datenbank ist (Campesato, 2023, S.260-261).

Mit Hilfe der SQL-Alchemy-Bibliothek wird im Python-Programm die Verbindung zur SQLite-Datenbank hergestellt, eine Tabelle angelegt und die Daten in die Datenbank überschrieben. Das Ergebnis ist abschließend jeweils eine lokale SQLite-Datei für den Trainings-, Ideal- und Testdatensatz.

2.4 UnitTest-Bibliothek

Das Testen des Python-Programms ist ein wichtiger Schritt bei der Entwicklung eines Python-Codes. Es wird sichergestellt, dass die Funktionalität des Programms wie erwartet gegeben ist. Fehler können frühzeitig aufgedeckt und behoben werden. Eine Möglichkeit, das Python-Programm wie beschrieben zu testen, bietet die UnitTest-Bibliothek. Sie ist einer der beliebtesten Test-Bibliotheken und gehört zu den Standardbibliotheken von Python (Kapil, 2019, S.238-240).

Mit der UnitTest-Bibliothek wird im vorliegenden Python-Programm die Funktionalität der Module der einzelnen Klassen überprüft. Es wird geprüft, ob Berechnungen korrekt durchgeführt werden und ob Attribute der Klassen richtig zurückgegeben werden. Außerdem wird das Anlegen von Instanzen der erzeugten Klassen überprüft.

2.5 Tkinter-Modul

Das Tkinter-Modul ist eine Schnittstelle zur Tk-Bibliothek, die zu den Standardbibliotheken von Python gehört. Durch das Tkinter-Modul ist es möglich, grafische Oberflächen für kleinere Anwendungen zu erstellen (Theis, 2017, S.371).

In dem Python-Programm wird das Modul filedialog des Pakets Tkinter verwendet, das vorgefertigte Dateidialoge bereitstellt. Die Dateidialoge fordern den Nutzer des Python-Programms dazu auf, Dateien oder Ordner auszuwählen, um diese im Python-Programm hineinzuladen (Ernesti & Kaiser, 2020, S.869).

Das Modul filedialog kommt in dieser Hausarbeit jedoch nur dann zum Einsatz, wenn die als Default eingebetteten Dateipfade nicht existieren. Um einen Abbruch des Programms zu verhindern, müssen schließlich die drei CSV-Dateien manuell ausgewählt werden.

2.6 OS-Modul

Das os-Modul ermöglicht es, betriebssystemabhängige Funktionalitäten zu nutzen (Python Software Foundation, 2024). Im erstellten Python-Programm werden mit Hilfe des os-Moduls die Datei-Pfade der zu importierenden CSV-Dateien überprüft. Existieren die als Default implementierten Dateipfade nicht, wird der Benutzer aufgefordert, die drei CSV-Dateien manuell auszuwählen.

2.7 Math-Modul

Durch das math-Modul erlangt der Anwender Zugang zu mathematischen Funktionen. Hierzu zählen unter anderem die trigonometrischen Funktionen (Theis, 2017, S.89-91). Für die Berechnung des Testkriteriums in Aufgabenteil 2 wird die Berechnung der Quadratwurzel benötigt, die ebenfalls eine Funktion des math-Moduls ist.

3 Struktur des Python-Programms

Dieses Kapitel behandelt die Struktur des entwickelten Python-Programms. Die Aufgabenstellung gibt vor, dass eine objektorientierte Programmierung angewendet werden soll. Außerdem sollen sowohl Standard- als auch user-definierte Exception-Handlings sinnvoll verwendet werden und Unit-Tests, wo immer es sich anbietet, implementiert werden (siehe Kapitel 2.4). In den folgenden Unterkapiteln werden die genannten Anforderungen vertieft und die Umsetzung im Python-Code erläutert.

3.1 Objektorientierte Programmierung

Unter einer objektorientierten Programmierung ist zu verstehen, dass alle Elemente, mit denen ein Python-Programm erstellt wird, Instanzen einer Klasse sind. Dies betrifft einzelne Zahlen genauso wie Listen, Dictionaries usw. (Ernesti & Kaiser, 2020, S.381).

Die Grundlage der objektorientierten Programmierung sind sogenannte Klassen, die mit Eigenschaften (Attribute) und Methoden (Funktionen) ausgestattet werden. Es können nun im Programm-Code mehrere Objekte (Instanzen) dieser Klassen erzeugt werden und die entsprechenden Methoden auf die Objekte angewendet werden. Weiterhin besteht die Möglichkeit, dass einzelne Klassen von einer anderen Klasse erben. Es wird eine Basisklasse (Eltern-Klasse) definiert, die ihre Eigenschaften und Methoden an die abgeleitete Klasse (Kind-Klasse) vererbt. Ähnliche Objekte mit identischen Eigenschaften und Methoden können so vereinfacht erzeugt werden (Theis, 2017, S.203).

Python ermöglicht es dem Anwender, bei der Definition der Attribute und Methoden einer Klasse aus drei Sichtbarkeitstufen zu wählen. Auf Attribute und Methoden, die als public deklariert werden, kann von außerhalb der Klasse ohne Einschränkungen zugegriffen werden. Ähnlich verhält es sich bei der Sichtbarkeitsstufe protected, die häufig in Vererbungshierachien Anwendung findet. Prinzipiell ist der Zugriff auf die Attribute und Methoden von außen uneingeschränkt möglich. Per Konvention wird jedoch festgelegt, dass die Sichtbarkeitsstufe protected wie die Sichtbarkeitsstufe privat behandelt wird. Hierbei ist kein Zugriff auf Attribute und Methoden von außerhalb möglich (Steyer, 2018, S.161).

In dieser Hausarbeit werden zur Bearbeitung der Aufgabenstellung zwei Basisklassen erzeugt, wobei eine Basisklasse drei abgeleitete Klassen besitzt. Zusätzlich sind in dem Python-Programm vier Klassen zum Abfangen von Fehlern (Exception-Handling) implementiert. Im Folgenden werden die zwei zur Lösung der Aufgabenstellung verwendeten Basis-Klassen inklusive ihrer abgeleiteten Klassen sowie ihre Funktionalitäten erläutert. Die erstellten Klassen des Python-Programms besitzen Attribute der Sichtbarkeitsstufe privat oder protected. Um auf diese Attribute zugreifen zu können, gibt es für jedes Attribut eine Getter- und Setter-Methode (Steyer, 2018, S.162-163). Diese Methoden beginnen im Python-Code jeweils mit hole (get) und setze (set). Um die folgenden Grafiken nicht zu überladen, werden sie nicht aufgeführt.

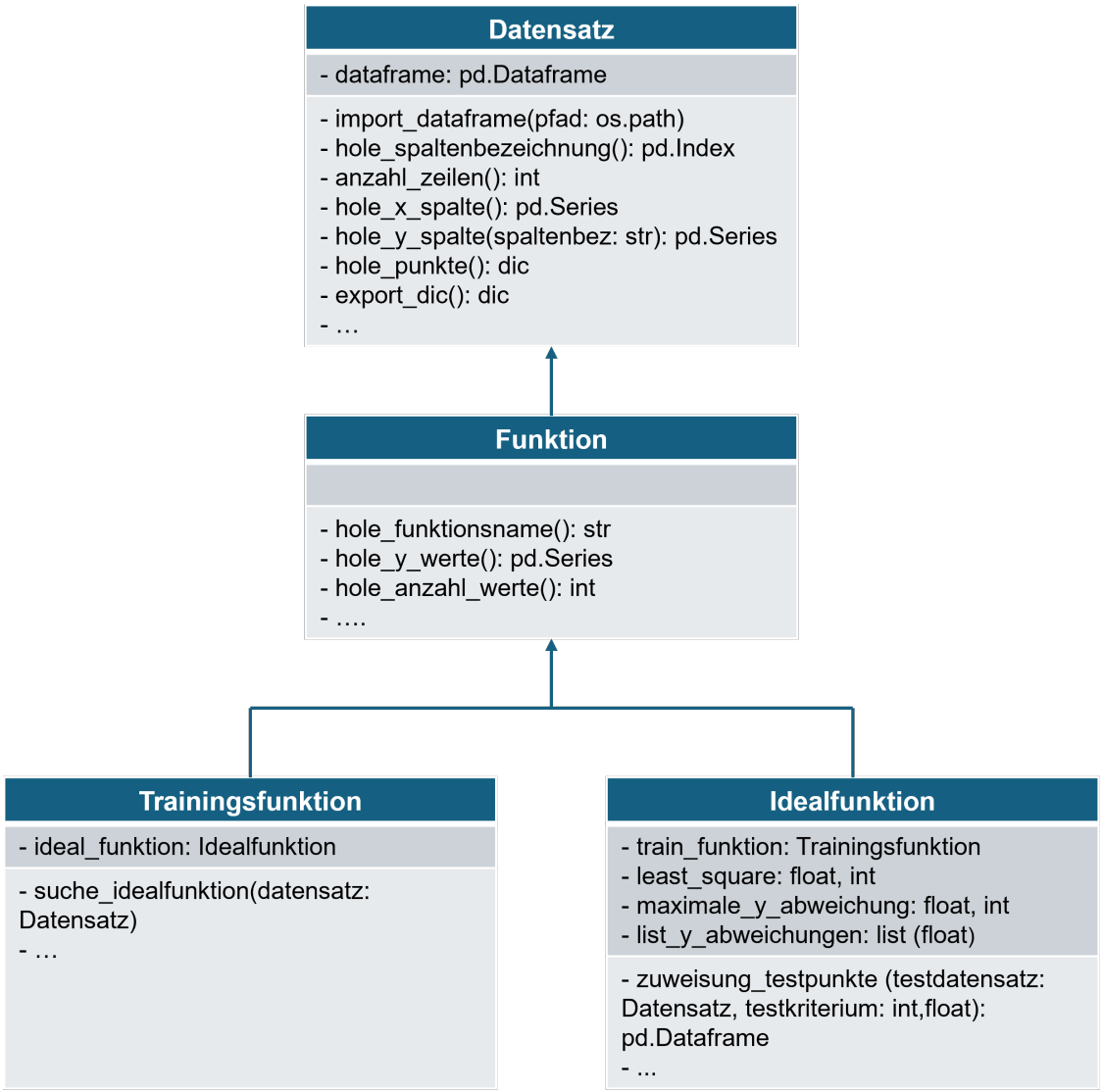
3.1.1 Klasse Datensatz

Die Klasse Datensatz bildet die Grundlage für die Verarbeitung der Daten aus den CSV-Dateien. Sie ist zudem eine Elternklasse der Klasse Funktion, die in Kapitel 3.1.2 erläutert wird. Abb. 3.1 zeigt die Attribute und Methoden der Klasse Datensatz sowie die entsprechenden Kindklassen.

Die Grafik zeigt, dass die Klasse Datensatz nur ein Attribut besitzt (dataframe). Bei der Erstellung der Klasse muss jedoch nicht zwingend eine Variable für das Attribut übertragen werden. In diesem Fall bekommt das Attribut dataframe den Wert einer leeren Zeichenkette zugewiesen. Somit kann eine Instanz der Klasse Datensatz entweder mit einer leeren Zeichenkette oder einem Pandas-Dataframe

(siehe Kapitel 2.1) erzeugt werden.

Abb. 3.1: Attribute und Methoden der Klasse Datensatz



Quelle: Eigene Darstellung

Wird einer Instanz der Klasse Datensatz bei der Erzeugung kein Pandas-Dataframe zugewiesen, bietet die Methode import_dataframe(pfad) die Möglichkeit, das Attribut dataframe durch den Import einer CSV-Datei zu belegen. Hierbei wird aus den Daten der CSV-Datei ein Pandas-Dataframe erzeugt und gespeichert.

Die in der folgenden Tabelle 3.1 beschriebenen Methoden können jeweils nur ausgeführt werden, wenn dem Attribut dataframe ein Pandas-Dataframe zugewiesen ist. Daher wird vor jeder Ausführung der Methoden unter Verwendung eines Exception-Handlings überprüft, ob es sich um ein Pandas-Dataframe oder eine leere Zeichenkette handelt.

Bei dem Großteil der in Tabelle 3.1 beschriebenen Methoden der Klasse Datensatz wird lediglich auf Funktionen des Datentyps Pandas-Dataframe zurückgegriffen. Die Klasse Datensatz dient nicht dazu, Berechnungen oder Ähnliches durchzuführen. Sie soll das Arbeiten mit einem Pandas-Dataframe erleichtern und anwenderfreundlicher gestalten. Die Bezeichnungen der Methoden sind auf die zu verarbeitenden Daten dieser Hausarbeit abgestimmt. Somit kann es bei der Anwendung der Klasse Datensatz

auf abweichende Datenstrukturen zu Verständnisproblemen kommen. Es ist z.B. zwingend notwendig, dass das Pandas-Dataframe in der ersten Spalte die X-Werte der Funktionen beinhaltet, um bei der Anwendung der Methode hole_x_spalte() ein korrekten Rückgabewert zu erhalten.

Tab. 3.1: Methoden der Klasse Datensatz

Methode	**Funktionalität**
hole_spaltenbezeichnung()	Die Spaltenbezeichnungen des Pandas-Dataframes werden ausgegeben (z.B. x,y1,...).
anzahl_zeilen()	Die Anzahl der Zeilen des Pandas-Dataframes wird zurückgegeben. Die Anzahl der Zeilen ist äquivalent zu der Anzahl der X- und Y-Werte der Funktionen.
hole_x_spalte()	Die X-Werte der Funktionen werden zurückgegeben.
hole_y_spalte(spaltenbez)	Die Y-Werte einer bestimmten Spalte (Funktion) werden zurückgegeben.
hole_punkte()	Die Punkte (X,Y) der einzelnen Funktionen werden zurückgegeben.
export_dic()	Das Pandas-Dataframe wird als Dictionary zurückgegeben.

Quelle: Eigene Darstellung

Zum einen wird die Klasse Datensatz verwendet, um die Daten aus den gegebenen CSV-Dateien zu importieren. Durch die implementierten Methoden werden die Daten gespeichert und für weitere Berechnungen aufbereitet.

Zum anderen werden die Funktionalitäten des Pandas-Dataframes, und damit auch der Klasse Datensatz, genutzt, um die Daten mit geringem Aufwand in eine SQLite-Datenbank zu exportieren.

3.1.2 Kindklasse Funktion

Die Klasse Funktion ist eine Kindklasse der Klasse Datensatz. Sie besitzt wiederum mit den Klassen Trainings- und Idealfunktion zwei abgeleitete Klassen (siehe Abb. 3.1). Mit Hilfe dieser Klassen werden die zur Lösung der Aufgabenstellung benötigten Berechnungen durchgeführt.

Die Klasse Funktion besitzt, neben dem Attribut dataframe aus der Elternklasse, kein weiteres Attribut. Jedoch ist es bei der Erstellung einer Instanz der Klasse Funktion zwingend erforderlich, ein Pandas-Dataframe zu übergeben. Wird kein Pandas-Dataframe übergeben, wird das Programm mit Hilfe des Exception-Handlings abgebrochen. Die Klasse Funktion verarbeitet nur die ersten beiden Spalten des Pandas-Dataframes, da hier per Definition die X- sowie Y-Werte der Funktion zu finden sind. Tabelle 3.2 zeigt die zur Klasse Funktion gehörenden Methoden und beschreibt ihre Funktionalität.

Tab. 3.2: Methoden der Klasse Funktion

Methode	**Funktionalität**
hole_funktionsname()	Der Funktionsname des Pandas-Dataframes (Index der zweiten Spalte) wird zurückgegeben (z.B. y1).
hole_y_werte()	Die Y-Werte der Funktion (zweite Spalte) werden zurückgegeben.

Quelle: Eigene Darstellung

Da bei der Erzeugung einer Instanz der Klasse Funktion die Konstruktor-Methode der Elternklasse Datensatz aufgerufen wird, ist die Grundlage der Klasse Funktion ebenfalls ein Pandas-Dataframe (siehe Abb. 3.1). Somit kann mit der Klasse Funktion problemlos auf die Methoden der Elternklasse

Datensatz zugegriffen werden. Die Funktionalitäten aller in Tabelle 3.2 gezeigten Methoden basieren auf Funktionen eines Pandas-Dataframes. Auch in diesem Fall stellen die Bezeichnungen der Methoden der Klasse Funktion eine Beziehung zu der Datenstruktur der vorgegebenen CSV-Dateien her.

Klasse Trainingsfunktion

Die Klasse Trainingsfunktion beinhaltet mit der Methode suche_idealfunktion(datensatz) die eigentliche Methode zur Bearbeitung von Aufgabenteil 1 (siehe Kapitel 1.1). Einer Trainingsfunktion wird bei der Anwendung dieser Methode ein Datensatz, also ein Pandas-Dataframe, zur weiteren Verarbeitung zugewiesen. Im Regelfall ist dies der Datensatz der Idealfunktionen. Es wird über alle Funktionen des Pandas-Dataframes der Idealfunktionen iteriert. Pro Spalte wird eine Vergleichsfunktion angelegt, die eine Instanz der Elternklasse Funktion ist. Sie dient lediglich als Vergleichsobjekt. Hierzu werden Methoden der Klasse Funktion, bspw. hole_y_werte(), benötigt. Nun wird über die Y-Werte der Vergleichsfunktion iteriert und jeweils die Y-Abweichung zum entsprechenden Y-Wert der Trainingsfunktion berechnet. Die Y-Abweichungen werden quadriert und summiert, um den Least-Square-Wert zu berechnen (siehe Formel 1.1). Die Vergleichsfunktion mit dem geringsten Least-Square-Wert wird schließlich als Instanz der Klasse Idealfunktion unter dem Attribut ideal_funktion gespeichert. Zusätzlich wird zur Anwendung des Testkriteriums für Aufgabenteil 2 (siehe Kapitel 1.2) die maximale Abweichung zwischen einem Y-Wert der Trainingsfunktion und dem entsprechenden Y-Wert der Idealfunktion bestimmt.

Klasse Idealfunktion

Instanzen der Klasse Idealfunktion dienen zum einen als Speicherort der bestimmten Funktionen zur entsprechenden Trainingsfunktion. Die Klasse Idealfunktion beinhaltet neben dem Attribut der Elternklasse (dataframe) vier weitere Attribute. Bei der Erzeugung einer Instanz der Klasse Idealfunktion müssen neben einem Pandas-Dataframe der Idealfunktion zusätzlich folgende Attribute zugewiesen werden (siehe Abb. 3.1):

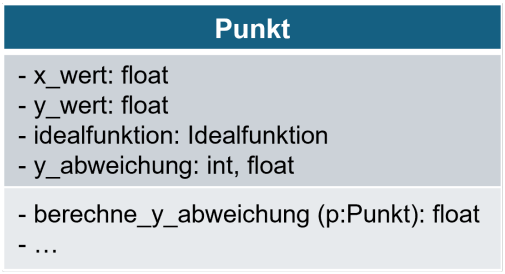
- Least-Square-Wert
- maximale Y-Abweichung
- Liste alle Y-Abweichungen zur Trainingsfunktion
- Trainingsfunktion

Zum anderen wird mit Hilfe der Methode zuweisung_testpunkte(testdatensatz, testkriterium) die Berechnung für Aufgabenteil 2 (siehe Kapitel 1.2) durchgeführt. Es wird der Idealfunktion der Testdatensatz (Instanz der Klasse Datensatz) inklusive Testkriterium zugewiesen. Anschließend wird über die Punkte des Testdatensatzes iteriert und für jeden Punkt überprüft, ob er zur Idealfunktion passt. Hierfür findet das Testkriterium Anwendung. Die Aufgabenstellung besagt, dass die Abweichung zwischen Testpunkt und Idealfunktion nicht größer sein darf als die maximale Abweichung zwischen Idealfunktion und zugehöriger Trainingsfunktion. In diesem Fall wird die maximale Abweichung zusätzlich mit dem Faktor $\sqrt{2}$ multipliziert, sodass das Testkriterium höher ist als die maximale Abweichung zwischen Ideal- und Trainingsfunktion. Passt ein Testpunkt zur Idealfunktion, wird er einem Pandas-Dataframe gemäß Aufgabenstellung zugewiesen. Nachdem für alle Testpunkte geprüft wurde, ob sie zu einer Idealfunktion passen, wird der Pandas-Dataframe zurückgegeben.

3.1.3 Klasse Punkt

Die Klasse Punkt ist definiert durch einen X- und einen Y-Wert. Mit Hilfe der Klasse Punkt werden die XY-Paare der einzelnen Funktionen verarbeitet und entsprechende Berechnungen durchgeführt. Abb. 3.2 zeigt die Attribute und Methoden der Klasse Punkt. Die Klasse Punkt ist eine alleinstehende Klasse, die von keiner weiteren Klasse erbt bzw. an eine weitere Klasse vererbt.

Abb. 3.2: Attribute und Methoden der Klasse Punkt



Quelle: Eigene Darstellung

Die Grundlage der Klasse Punkt ist der X- und Y-Wert. Bei der Erzeugung einer Instanz der Klasse Punkt müssen die beiden Werte zwingend als Attribute übergeben werden.

Die Klasse Punkt findet unter anderem Anwendung beim Export der Punkte aus der Klasse Datensatz/Funktion (siehe Methode hole_punkte(), Kapitel 3.1). Außerdem wird die Methode berechne_y_abweichung(p) verwendet, um bei der Berechnung für Aufgabenteil 2 die Y-Abweichung zwischen zwei Punkten zu ermitteln. Neben den beiden Attributen x_wert und y_wert besitzt die Klasse Punkt zwei weitere Attribute. Zum einen kann dem Punkt mit dem Attribut idealfunktion eine Idealfunktion (Instanz der Klasse Idealfunktion) zugewiesen werden. Zum anderen kann die entsprechende Y-Abweichung zwischen dem Testpunkt und der Idealfunktion unter dem Attribut y_abweichung hinterlegt werden.

3.2 Exception-Handling

Das Exception-Handling bzw. die Ausnahmebehandlung beschreibt das Bereinigen von Programmfehlern, die auf äußere Umstände zurückzuführen sind. Unter einem Programmfehler (Ausnahme) ist eine Unterbrechung des normalen Programmablaufs zu verstehen. Ohne Behebung des Programmfehlers kann der Programmablauf nicht fortgesetzt werden. Umgesetzt wird das Exception-Handling, indem im Fall eines Programmfehlers eine Behandlungsmaßnahme anstelle des eigentlich vorgesehenen Programmcodes ausgeführt wird. Realisiert wird dieses Verfahren in Python durch sogenannte try-except-Blöcke. Der try-Block beinhaltet den Programmcode, der mit dem Exception-Handling behandelt werden soll. Tritt im try-Block eine Ausnahme auf, wird der except-Block ausgeführt. Das Programm wird hierbei nicht beendet, sondern es wird der Programmcode im except-Block ausgeführt. Python bietet eine Reihe von Standard-Exceptions, die eine qualifizierte Reaktion je nach Ausnahmetyp ermöglichen.

Außerdem bietet Python die Möglichkeit, mit der raise-Anweisung benutzerdefinierten Ausnahmeklassen zu aktivieren (Steyer, 2018, S.181-193).

In dem vorliegenden Python-Programm werden sowohl Standard-Ausnahmen als auch benutzerdefinierte Ausnahmen verwendet. Die implementierten benutzerdefinierte Ausnahmen können Tabelle 3.3 entnommen werden.

Tab. 3.3: Benutzerdefinierte Ausnahmen

Exception-Klasse	**Funktionalität**
DatensatzError()	Abfangen von Ausnahmen in der Klasse Datensatz.
FunktionError()	Abfangen von Ausnahmen in der Klasse Funktion.
PunktError()	Abfangen von Ausnahmen in der Klasse Punkt.
CSVError()	Abfangen von Ausnahmen beim Import einer CSV-Datei.

Quelle: Eigene Darstellung

Es ist zu erkennen, dass es für jede Klasse eine eigene Ausnahmebehandlung gibt, die in der Konsole ausgegeben wird. So wird beim Blick in die Konsole sofort deutlich, in welcher Klasse eine Ausnahme aufgetreten ist.

Da bei der Erzeugung einer Instanz der Klasse Datensatz nicht zwingend ein Pandas-Dataframe übertragen werden muss (siehe Kapitel 3.1), findet das Exception-Handling hier bei jeder Methode Anwendung. Das ist notwendig, da die meisten Methoden der Klasse Datensatz ein Pandas-Dataframe als Grundlage benötigen. Ist jedoch kein Pandas-Dataframe hinterlegt, können die Methoden nicht korrekt ausgeführt werden. Es werden die Ausnahmen abgefangen und in der Konsole ausgegeben. Da ein weiterer Ablauf des Programms in diesen Fällen nicht möglich wäre, wird das Programm bei der Ausnahmebehandlung abgebrochen. So kann der Anwender direkt die Fehlerart und die Stelle, an der die Ausnahme auftritt, erkennen.

Da bei er Erzeugung einer Instanz der Klasse Funktion zwingend ein Pandas-Dataframe zugewiesen werden muss (siehe Kapitel 3.1.2), ist keine Ausnahmebehandlung bei den Methoden notwendig, die lediglich Daten des Pandas-Dataframe zurückgeben. Weitere Ausnahmebehandlung finden Anwendung bei Methoden zur Berechnung der Idealfunktion bzw. der Passung der Testpunkte. Außerdem wird bei jedem Setzen von Variablen über klassenspezifische Methoden geprüft, ob die zugewiesene Variable den korrekten Datentyp besitzt.

Bei der Klasse Punkt (siehe Kapitel 3.1.3) verhält es sich ähnlich wie bei der Klasse Funktion. Auch bei der Erzeugung einer Instanz der Klasse Punkt wird überprüft, ob die zugewiesenen Variablen einen korrekten Datentyp besitzen.

Eine Besonderheit bei den benutzerdefinierten Ausnahmebehandlungen ist der CSVError. Ist der als Default hinterlegte Dateipfad oder der vom Nutzer ausgewählte Dateipfad keine CSV-Datei, wird der CSV-Error ausgelöst. Der Anwender erkennt nun in der Python-Konsole, dass es Probleme mit dem Import einer CSV-Datei gibt und kann entsprechend reagieren.

Neben den benutzerdefinierten Exception-Handlings werden im vorliegenden Python-Programm auch Standard-Exception-Handlings verwendet. Diese werden immer dann verwendet, wenn Attribute einer Klasse zurückgegeben werden sollen, die zuvor berechnet und zugewiesen werden müssen. Ein Beispiel hierfür ist der AttributeError beim Aufrufen der Methode hole_idealfunktion() der Klasse Trainingsfunktion (siehe Abb. 3.1). Das Attribut ideal_funktion wird erst durch das Aufrufen der Methode suche_idealfunktion(datensatz) belegt. Wird nun die Methode hole_idealfunktion() vor der Methode suche_idealfunktion(datensatz) aufgerufen, tritt ein Programmfehler auf. Dieser Programmfehler wird mittels des AttributeError abgefangen.

3.3 Anwendung von Unit-Tests

Wie in Kapitel 2.4 beschrieben wird die Funktionalität des vorliegenden Programms mittels Unit-Tests überprüft. Es werden im Wesentlichen drei unterschiedliche Funktionalitäten der konstruierten Klassen getestet.

Zum einen wird die jeweilige Konstruktor-Methode je Klasse überprüft. Mit Hilfe der Konstruktormethode (__init__) können einer Instanz einer Klasse ein oder mehrere Anfangswerte zugewiesen werden (Theis, 2017, S.206). Im Unit-Test-Modul werden Instanzen der drei Klassen (siehe Kapitel 3.1) bewusst falsche Anfangswerte übertragen. Es wird nun geprüft, ob die richtigen Fehlermeldungen ausgegeben werden. In diesem Fall handelt es sich um die benutzerdefinierten Ausnahmen.

Außerdem werden die Getter- und Setter-Methoden der Klassen geprüft. Hier wird geprüft, ob die Klassen die korrekten Werte der privaten bzw. geschützten Attribute zurückgeben.

Zuletzt werden die Methoden überprüft, die Berechnungen durchführen. Mit einfachen Werten werden die Rechenmethoden durchgeführt und geprüft, ob die Methoden das richtige Ergebnis zurückgeben.

4 Lösungen der Aufgabenstellungen

In den folgenden Kapiteln werden die Lösungen der Aufgabenstellungen präsentiert. Außerdem werden die Visualisierungen der Lösungen dargestellt und erklärt.

4.1 Passungen aus Trainings- und Idealfunktion

Das Ziel der Aufgabenstellung 1 ist es, den vier gegebenen Trainingsfunktion jeweils eine Idealfunktion zuzuweisen (siehe Kapitel 1.1). Folgende Tabelle zeigt die Passungen aus Trainings- und Idealfunktion.

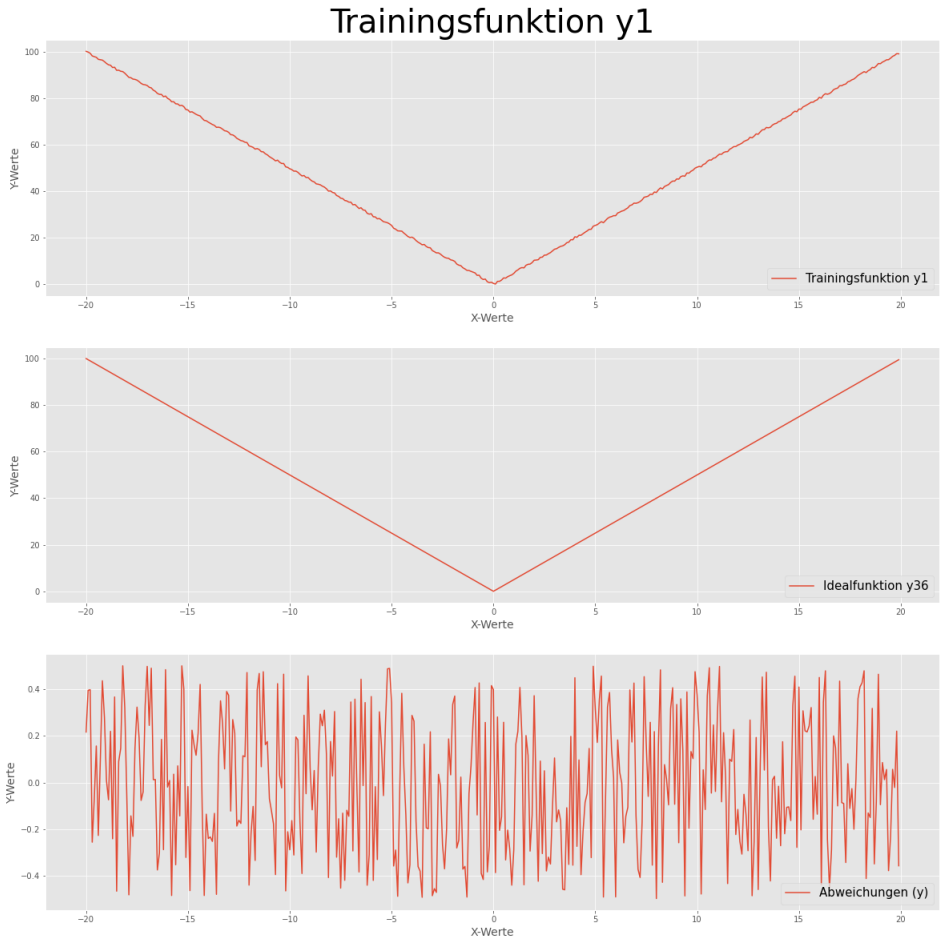
Tab. 4.1: Passungen aus Trainings- und Idealfunktion

Trainingsfunktion	Idealfunktion
Y1	Y36
Y2	Y11
Y3	Y2
Y4	Y33

Quelle:Eigene Darstellung

Die Visualisierung der Ergebnisse erfolgt durch das Abbilden der Trainings- sowie Idealfunktion untereinander. Zusätzlich zeigt die dritte Grafik die Y-Abweichungen zwischen Trainings- und Idealfunktion. Abb. 4.1 zeigt exemplarisch die Visualisierung für die Trainingsfunktion Y1.

Abb. 4.1: Visualisierung der Trainingsfunktion 1



Quelle: Eigene Darstellung

Es ist deutlich zu sehen, dass die Trainingsfunktion keine geradlinige Funktion ist. Sie besteht aus einer Vielzahl von Messungen, die von einer Idealfunktion leicht abweichen. Die Idealfunktion ist in der mittleren Grafik (siehe Abb. 4.1) dargstellt. Die Abweichungen der Trainingsfunktion von der Idealfunktion sind in der untersten Grafik dargstellt. Zu sehen ist, dass keine Abweichung größer als 0.5 bzw. kleiner als -0.5 ist.

Das Python-Programm erzeugt für jede Trainingsfunktion eine ähnliche Grafik, die im selben Ordner gespeichert wird, in dem sich die Python-Datei befindet.

4.2 Anpassbare Testpunkte

Die Aufgabenstellung 2 fordert das Zuweisen von den gegebenen Testpunkte zu einer oder mehreren Idealfunktionen. Hierfür ist ein Testkriterium vorgegeben, das die Testpunkte erfüllen müssen (siehe Kapitel 1.2). Jeder Idealfunktion kann mindestens ein Testpunkt zugewiesen werden. Tabelle 4.2 zeigt die Anzahl der Testpunkte je Idealfunktion.

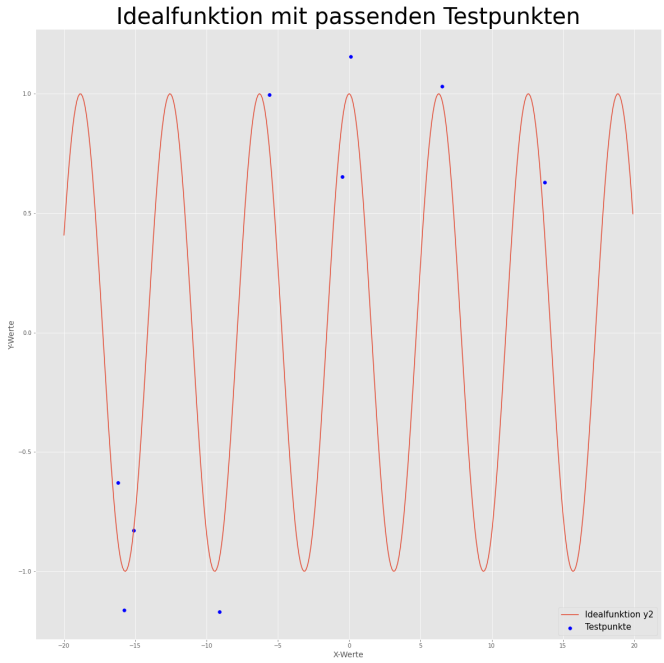
Tab. 4.2: Anzal Testpunkte pro Idealfunktion

Idealfunktion	Anzahl Testpunkte
Y2	9
Y11	11
Y33	11
Y36	4

Quelle: Eigene Darstellung

Die Visualisierung der Ergebnisse erfolgt durch das Darstellen der Idealfunktion als Graphen. In dieselbe Grafik sind die passenden Testpunkte als Punkte dargestellt, sodass ihre Abweichungen von der Idealfunktion grob abgegriffen werden können.

Abb. 4.2: Visualisierung der Idealfunktion Y2 und Testpunkte



Quelle: Eigene Darstellung

Abb. 4.2 zeigt die Visualisierung der Aufgabenstellung 2 exemplarisch für die Idealfunktion Y2. Die Abweichungen zwischen den Testpunkten und der Idealfunktion werden abgespeichert und später in eine entsprechende SQLite-Datei exportiert (siehe Kapitel 4.3). Für jede der vier Idealfunktionen wird eine nach demselben Muster erstellte Grafik (siehe Abb. 4.2) erzeugt und gespeichert.

4.3 SQLite-Export

Laut Aufgabenstellung sollen der Trainingsdatensatz, der Idealdatensatz und die Ergebnisse aus Aufgabenstellung 2 jeweils in eine SQLite-Datenbank (Datei) exportiert werden. Der Zugriff auf die SQLite-Datenbank erfolgt wie in Kapitel 2.3 beschrieben mit Hilfe der SQL-Alchemy-Bibliothek. Das vorliegende Python-Programm erzeugt hierfür drei separate SQLite-Dateien, die jeweils nur eine Tabelle enthalten.

Der Trainings- bzw. Idealdatensatz muss nicht weiter verändert werden und kann dementsprechend ohne weitere Bearbeitung exportiert werden. Die Ergebnisse der Aufgabenstellung 2 werden zum einfacheren Export in einer Instanz der Klasse Datensatz (siehe Kapitel 3.1) gespeichert. Somit liegen die Ergebnisse als Pandas-Dataframe vor und können exportiert werden. Abb. 4.3 zeigt die Struktur der Tabelle der resultierenden SQLite-Datenbank.

Abb. 4.3: Struktur der SQLite-Datenbank für die Testpunkte

X (Testfunktion)	Y (Testfunktion)	Delta Y	Idealfunktion
-2.8	13.655836	-0.34416399999999925	y36
0.5	2.4143674	-0.08563259999999984	y36
15.1	75.25758	-0.24241999999999564	y36
5.8	28.826637	-0.17336299999999838	y36
17.9	17.754583	-0.14541699999999835	y11
19.7	19.575151	-0.12484899999999755	y11

Quelle: Eigene Darstellung

5 Reflexion des Python-Programms

In diesem Kapitel wird das vorliegende Python-Programm, welches die Grundlage dieser Hausarbeit bildet, reflektiert. Es wird die verwendete Struktur erklärt und mögliche Alternativen dargelegt und erörtert.

Die am kritischsten zu hinterfragende Klasse im Programm-Code ist die Klasse Datensatz. Beim Erzeugen einer Instanz der Klasse Datensatz muss nicht zwingend eine Anfangswert übertragen werden (siehe Abb. 3.1). Das führt dazu, dass das Konstruieren sowie Anwenden der weiteren Methoden komplizierter ist, als wenn zwingend ein Pandas-Dataframe zugewiesen werden müsste. Bei jeder Anwendung einer Methode muss überprüft werden, ob dem Objekt ein Pandas-Dataframe zugewiesen wurde. Das führt dazu, dass die Anwendung der Klasse Datensatz fehleranfällig wird. Mit Hilfe des Exception-Handlings (siehe Kapitel 3.2) werden die zu erwartenden Ausnahmen jedoch gezielt abgefangen. Der Anwender bekommt in der Python-Konsole eine Fehlermitteilung und kann entsprechend reagieren.

Der Vorteil eines Konstruktors ohne zwingende Zuweisung eines Attributs ist in diesem Fall, dass die Instanz der Klasse Datensatz mit einer Methode durch eine CSV-Datei gefüllt werden kann. Es wird dem Anwender eine Möglichkeit geboten, den Pfad einer CSV-Datei zu übertragen, wobei anschließend aus den Daten der CSV-Datei ein Pandas-Dataframe erzeugt wird.

Weiterhin kann kritisiert werden, dass die Klasse Datensatz zum Großteil lediglich Methoden der Klasse Pandas-Dataframe beinhaltet. So kann der Eindruck entstehen, dass die Klasse Datensatz überflüssig ist. Alle Funktionen/Methoden könnten im normalen Programm-Code über die Funktionen eines Pandas-Dataframes ausgeführt werden. Die Klasse Datensatz hat jedoch in erster Linie den Auftrag, dem Anwender die Bearbeitung der Aufgabenstellung zu erleichtern. Sie ist eine aufgabenspezifische Klasse, welche die benötigten Funktionen des komplexen Pandas-Dataframes bündelt. Der Anwender muss sich nicht mit den Funktionen eines Pandas-Dataframes auseinandersetzen, sondern hat alle Methoden, die für die Lösung der Aufgabe notwendig sind, einfach formuliert vorliegen. Ein Beispiel hierfür ist die Methode anzahl_zeilen(). Dem Anwender ist sofort klar, welchen Wert diese Methode zurückgibt. Die entsprechende Funktion bzw. das Attribut des Pandas-Dataframes (dataframe.shape[0]) hingegen macht nicht deutlich, was zurückgegeben wird.

Außerdem wäre es möglich, auf die Klasse Funktion zu verzichten. Da ihre Grundlage ebenfalls ein Pandas-Dataframe ist, ähnelt sie sehr stark der Klasse Datensatz. Sie besitzen ähnliche Methoden und Attribute. Der entscheidende Unterschied ist jedoch, dass die Klasse Funktion nur die ersten zwei Spalten des Pandas-Dataframes verarbeitet. Somit werden bei der Klasse Funktion nur die Spalte mit den X-Werten und die Spalte mit den ersten Y-Werten berücksichtigt. Für die Klasse Funktion ergibt es nur Sinn, ein zweispaltiges Pandas-Dataframe als Anfangsattribut zu übertragen. Durch die Vererbungshierarchie können Methoden aus der Klasse Datensatz, die für beide Klassen notwendig sind, auch von der Klasse Funktion verwendet werden. Außerdem besitzen die beiden genannten Klassen einen weiteren Unterschied, der eine Trennung der Klassen durch Vererbung sinnvoll macht. Die Klasse Datensatz dient vielmehr zur Speicherung und Verarbeitung der Eingangs- und Ausgangsdaten. Die Klasse Funktion hingegen führt die elementaren Berechnungen und Vergleiche durch und bestimmt somit die Ergebnisse der Aufgabenstellungen.

Ein weitere Möglichkeit, den Programm-Code zu verkürzen, besteht darin, dass ausschließlich eine Klasse Funktion verwendet wird. Die Methoden und Attribute der abgeleiteten Klassen Trainings- und Idealfunktion könnten auch in der Klasse Funktion implementiert werden. Da die beiden abgeleiteten

Klassen jedoch jeweils eine Methode besitzen, die für die Lösung der Aufgabenstellung wichtig ist, ist hier eine strikte Unterscheidung sinnvoll. Es wird eine Trennung zwischen den beiden Funktionstypen geschaffen, da bspw. die Idealfunktion deutlich mehr Attribute als die Trainingsfunktion benötigt. Außerdem ist bei der Erzeugung einer Instanz der Klassen erkennbar, um welche Art von Funktion es sich handelt. Die Unterscheidung zwischen Trainings- und Idealfunktion kann bei nur einer Klasse Funktion alternativ durch die Bezeichnung/Variable der Instanz der Klasse Funktion verdeutlicht werden. In dem vorliegen Python-Programm ist jedoch die Übersicht bereits durch das Implementieren der beiden Klassen Trainings- und Idealfunktion gegeben.

Um den Gedankengang einer objektorientierten Programmierung strikter zu verfolgen, wäre es möglich gewesen, den Programm-Code, der sich außerhalb des Definitionsbereich der Klassen befindet, in die entsprechenden Klassen als Methode zu integrieren. Bei dem genannten Programm-Code handelt es sich um das Erzeugen der Grafiken und der Export der Daten in eine SQLite-Datei. Da die SQL-Alchemy- und Matplotlib-Bibliothek viele Parameter zur Einstellung der Visualisierung bzw. den Daten-Export besitzen, sind in dem vorliegenden Python-Programm keine entsprechenden Methoden vorzufinden. Es müssten der Methode zu viele Parameter zugewiesen werden, sodass die Übersichtlichkeit verloren geht. Einfacher ist es hier, die Parameter genau an der Stelle im Programm anzugeben, an denen auch die Erzeugung der Grafiken bzw. das Exportieren der Daten vollzogen wird.

Der Aufbau der Visualisierungen der Lösung aus Aufgabenteil 1 (siehe Abb. 4.1) folgt dem Gedanken der Übersichtlichkeit in dieser Hausarbeit. Alternativ könnten alle Grafiken (Trainingsfunktion, Idealfunktion, Y-Abweichungen) in einer Grafik/Darstellung vereint werden. Der entscheidende Nachteil ist dann, dass der Unterschied zwischen Trainings- und Idealfunktion kaum erkennbar ist. In den vorliegenden Visualisierungen ist deutlich zu erkennen, dass die Trainingsfunktion eine Reihe von Messwerten und die Idealfunktion eine geradlinige Funktion darstellt. Die dargestellten Y-Abweichungen besitzen entlang der Y-Achse einen anderen Maßstab als die Trainings- sowie Idealfunktion. So können die Abweichungen deutlicher dargestellt werden und Ausreißer schnell ausfindig gemacht werden.

6 Reflexion der Aufgabenstellung

Dieses Kapitel reflektiert die Aufgabenstellung und stellt dar, ob und inwiefern das Python-Programm für von der Hausarbeit abweichende Anforderungen verwendet werden kann.
Die vorliegende Aufgabenstellung ist stark abhängig von den vorliegenden Datensätzen (CSV-Dateien). Somit hängt auch das erstellte Python-Programm sehr stark von der Struktur der CSV-Dateien ab. Das bedeutet, dass das Python-Programm fehlerfrei funktioniert, wenn andere Datensätze mit identischer Datenstruktur verwendet werden. Somit kann es für das Python-Programm auch eine Anwendung außerhalb dieser Hausarbeit geben, wenn die Vorgaben für die Datenstrukturen eingehalten werden. Hierbei ist die Anzahl der Trainings- und Idealfunktionen sowie der Testpunkte irrelevant. Für die Berechnungen ist es jedoch wichtig, dass die Trainings- und Idealfunktionen die gleiche Anzahl an Werten je Funktion besitzen. Außerdem müssen sie die identischen X-Werte besitzen, um sie miteinander vergleichen zu können. Somit müssen die Datensätze für die Ideal- bzw. Trainingsfunktionen die gleiche Anzahl an Zeilen besitzen, die Zahl der Spalten (Zahl der Funktionen) ist jedoch nebensächlich.

Weiterhin liegt für Aufgabenteil 1 kein Testkriterium vor, das besagt, ab welchem Least-Square-Wert eine Idealfunktion zu einer Testfunktion passt. Vielmehr ist von Beginn an klar, dass es für jede Trainingsfunktion eine sehr gut passende Idealfunktion gibt. Werden nun Datensätze verwendet, bei denen es zu einer Trainingsfunktion keine passende Idealfunktion gibt, bestimmt das Python-Programm trotzdem eine Idealfunktion. Der Trainingsfunktion würde eine Idealfunktion mit dem kleinsten Least-Square-Wert zugewiesen, auch wenn dieser sehr hoch ist. Somit würde es Sinn ergeben, für Aufgabenteil 1 ein Testkriterium festzulegen, das definiert, ab welchem Least-Square-Wert eine Idealfunktion zu einer Trainingsfunktion passt. Außerdem führt das Zuweisen eine Funktion zu einer Trainingsfunktion, die einen hohen Least-Square-Wert besitzt, dazu, dass der 'Idealfunktion' viele Testpunkte zugewiesen werden. Das Testkriterium in Aufgabenteil 2 hängt von den Abweichungen zwischen der Trainings- und Idealfunktion ab. Sind diese Abweichungen groß, kann folglich auch die Abweichung zwischen Testpunkt und Idealfunktion groß sein. Alle genannten Punkte führen dazu, dass ein Testkriterium für die Zuweisung einer Idealfunktion zu einer Trainingsfunktion sinnvoll ist.

Alles in allem ist das vorliegende Python-Programm in der Lage, die vorliegenden Datensätze und Datensätze mit identischer Datenstruktur zu verarbeiten. Es besitzt Ausnahmebehandlungen, um die Robustheit des Programms zu gewährleisten. Die objektorientierte Programmierung gewährleistet die Übersichtlichkeit des Programms und erlaubt es, einzelne Methoden in den Klassen nachträglich zu implementieren. Somit kann das Python-Programm für ähnliche Anforderungen erweitert werden.

Literaturverzeichnis

Campesato, O.(2023). *Bash for Data scientists.* Mercury Learning and inforMation LLc.

Ernesti, J. & Kaiser, P (2020). *Python 3: Das umfassende Handbuch* (6. Aufl.). Rheinwerk Verlag.

Kapil, S. (2019). *Clean Python : Elegant Coding in Python.* Apress Media LLC.

Matthes, E. (2019). *Python Crash Course, 2nd Edition : A Hands-On, Project-Based Introduction to Programming.* No Starch Press.

McKinney, W. (2023). *Datenanalyse Mit Python: Auswertung Von Daten Mit Pandas, NumPy und Jupyter* (3. Aufl.). O'Reilly Verlag GmbH & Co. KG.

Python Software Foundation (2024). *os.* https://docs.python.org/3/library/os.html

Theis, T. (2017). *Einstieg in Python*(5. Aufl.). Rheinwerk Verlag.

VanderPlas, J. (2018) .*Data Science mit Python : Das Handbuch für den Einsatz von IPython, Jupyter, NumPy, Pandas, Matplotlib und Scikit-Learn* (1. Aufl.). mitp Verlags GmbH & Co. KG.

Steyer, R. (2018). *Programmierung in Python. Ein kompakter Einstieg in die Praxis.* Springer Vieweg.

Anhang

A.1 Python-Programm

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 25 16:58:25 2024

@author: bauma
"""
import pandas as pd
import os.path
import os
import math as math
from matplotlib import pyplot as plt
from matplotlib import style
from sqlalchemy import create_engine
from sqlalchemy import text
import sqlalchemy as db
from tkinter import *
from tkinter import filedialog

class FunktionError (Exception):

    '''
    Diese Klasse ermöglicht das gezielte Abfangen von Fehlern in der Klasse
    Funktion und die Ausgabe in der Konsole.
    '''
    def __init__(self,message=""):
        self.funktion_message = 'Es wurde ein falscher Datentyp zur Erstellung\
        einer Funktion übergeben. Es muss ein Panda-Dataframe übergeben werden!'


class DatensatzError (Exception):
    '''
    Diese Klasse ermöglicht das gezielte Abfangen von Fehlern in der Klasse
    Datensatz und die Ausgabe in der Konsole.
    '''
    def __init__(self,message=""):

        self.datensatz_message = 'Es wurde dem Datensatz kein Dataframe\
        zugewiesen.'
        self.datensatz_funktion_message = "Zur Bestimmung der Idealfunktion\
        muss eine Instanz der Klasse Datensatz übergeben werden!"

class CSVError (Exception):
    '''
    Diese Klasse ermöglicht das gezielte Abfangen von Fehlern beim Import
    eines Datensatzes im CSV-Format.
    '''
    def __init__(self):
        self.csv_message = 'Es wurde keine CSV-Datei ausgewählt!'

class PunktError(Exception):
    '''
    Diese Klasse ermöglicht das gezielte Abfangen von Fehlern in der Klasse
    Punkt und die Ausgabe in der Konsole.
    '''
    def __init__(self,message=""):
        self.punkt_message = 'Mindestens einer der Variablen ist keine Zahl\
        (Float oder Integer)'
        self.punkt_instanz_message = 'Die zugewiesene Variable ist keine\
        Instanz der Klasse Punkt'

class Datensatz():
```

1

```
'''
Diese Klasse hat als Grundlage einen Pandas-Dataframe. Sie vereinfacht das
Arbeiten mit dem Dataframe bezogen auf die Struktur der zu importierenden
CSV-Dateien (x,y1,y2,...)
'''
def __init__(self, dataframe=""):
    '''
    Diese Methode wird beim Erstellen einer Instanz der Klasse Datensatz
    ausgeführt. Da es verschiedene Möglichkeiten gibt, der Klasse Datensatz
    einen Pandas-Datensatz zu übergeben, muss nicht zwingend bei der
    Erstellung der Klasse ein Dataframe übergeben werden.

    Input Argument:
    - dataframe = Pandas Dataframe oder leere Zeichenkette

    Output Argument:
    - keine

    Bei der Erstellung der Instanz wird überprüft, ob der übergebene Wert
    ein Dataframe oder eine leere Zeichenkette ist. Bei allen anderen Typen
    wird ein DatensatzError hervorgerufen.
    '''

    #Prüfen, ob es sich um einen Pandas Dataframe (Fall 1) handelt
    if isinstance(dataframe, pd.DataFrame):
        self._dataframe = dataframe

    #Prüfen, ob es sich um eine leere Zeichenkette (Fall 2) handelt
    elif dataframe == "":
        self._dataframe = dataframe

    #Treten Fall 1 und Fall 2 nicht ein --> DatensatzError
    else:
        print('ert')
        raise DatensatzError ('Es wurde dem Datensatz kein Dataframe \
        zugewiesen.')

def import_dataframe(self, pfad):
    '''
    Diese Methode erlaubt das Hinzufügen eines Dataframes durch das
    Importieren aus einer CSV-Datei.

    Input-Argument:
    - Dateipfad

    Output-Argument
    - keine
    '''
    #Prüfen, ob die Datei aus dem Pfad existiert
    if os.path.isfile(pfad):
        #Prüfen, ob die Datei eine CSV-Datei ist
        if pfad.endswith('.csv'):
            self._dataframe = pd.read_csv(pfad)
        #Ist die Datei keine CSV-Datei --> CSVError
        else:

            raise CSVError (CSVError().csv_message)


    #Wenn die Datei nicht existiert --> Manuelles Auswählen der
    #entsprechenden Dateien
```

2

```
        else:
            #Auswahl Trainingsfunktionen
            if pfad.endswith('train.csv'):
                titel = 'Bitte Trainingsfunktionen auswählen!'
            #Auswahl Idealfunktionen
            elif pfad.endswith('ideal.csv'):
                titel = 'Bitte Idealfunktionen auswählen!'
            #Auswahl Testpunkte
            elif pfad.endswith('test.csv'):
                titel = 'Bitte Testpunkte auswählen!'
            else:
                titel = 'Bitte Datei auswählen'

            while True:

                try:

                    pfad = filedialog.askopenfilename(title = titel,filetypes =\
                    (('CSV Files','*.csv'),('All Files','*.*')))
                    if pfad.endswith('.csv'):
                        self._dataframe = pd.read_csv(pfad)
                        break
                    else:
                        raise CSVError

                except CSVError:
                    print (CSVError().csv_message)


    def setze_dataframe (self, dataframe):
        '''
        Diese Methode ermöglicht die Übergabe eines Pandas-Dataframe an
        die erstellte Instanz der Klasse Datensatz.

        Input-Argument:
        - dataframe = Pandas Dataframe

        Output-Argument:
        - keine
        '''

        try:
            #Prüfen, ob dataframe ein Pandas Dataframe ist
            if isinstance(dataframe, pd.DataFrame):
                self._dataframe = dataframe
            #Ist die Variable Dataframe keine Instanz der Klasse Dataframe
            #--> DatensatzError
            else:
                raise DatensatzError

        except DatensatzError:
            print (DatensatzError().datensatz_message)

    def hole_dataframe (self):
        '''
        Diese Methode gibt den Pandas Dataframes des Datensatzes zurück.

        Input-Argument:
        -keine

        Output-Argument:
```

3

```
        - self.__dataframe (Instanz der Klasse Pandas DataFrame)
        '''
        #Prüfen, ob dem Datensatz bereits ein Dataframe zugewiesen wurde
        if isinstance(self._dataframe,pd.DataFrame):
            return self._dataframe

        else:
            raise DatensatzError('Dem Datensatz wurde kein Dataframe zugewiesen!')



    def hole_spaltenbezeichnung(self):
        '''
        Diese Methode gibt die Spaltenbezeichnungen des Dataframes zurürck.

        Input-Argument:
        -keine

        Output-Argument:
        - self.__dataframe.columns (Spaltenbezeichnungen)
        '''
        #Prüfen, ob dem Datensatz bereits ein Dataframe zugewiesen wurde
        if isinstance(self._dataframe,pd.DataFrame):
            return self._dataframe.columns

        else:
            raise DatensatzError('Dem Datensatz wurde kein Dataframe zugewiesen!')

    def anzahl_zeilen(self):
        '''
        Diese Methode gibt die Anzahl der Zeilen des Dataframes zurürck.

        Input-Argument:
        -keine

        Output-Argument:
        - self.__dataframe.shape[0] (Anzahl der Zeilen)
        '''
        #Prüfen, ob dem Datensatz bereits ein Dataframe zugewiesen wurde
        if isinstance(self._dataframe,pd.DataFrame):
            return self._dataframe.shape[0]

        else:
            raise DatensatzError('Dem Datensatz wurde kein Dataframe zugewiesen!')

    def hole_x_spalte (self):
        '''
        Diese Methode gibt die X-Spalte (X-Werte) der Funktionen des Dataframes
        zurürck.

        Input-Argument:
        -keine

        Output-Argument:
        - slef.__dataframe['x'] (X-Werte der Funktionen)
        '''
        #Prüfen, ob dem Datensatz bereits ein Dataframe zugewiesen wurde
        if isinstance(self._dataframe,pd.DataFrame):
            return self._dataframe['x']

        else:
```

4

```
            raise DatensatzError('Dem Datensatz wurde kein Dataframe zugewiesen!')

    def hole_y_spalte (self, spaltenbez):
        '''
        Diese Methode gibt die Y-Spalte (Y-Werte) der Funktionen des Dataframes
        zurürck.

        Input-Argument:
        -spaltenbez (Spaltenbezeichnung, die zurückgegeben werden soll)

        Output-Argument:
        - self.__dataframe[spaltenbez] (Y-Werte der Funktionen)
        '''
        #Prüfen, ob dem Datensatz bereits ein Dataframe zugewiesen wurde
        if isinstance(self._dataframe,pd.DataFrame):
            return self._dataframe[spaltenbez]

        else:
            raise DatensatzError('Dem Datensatz wurde kein Dataframe zugewiesen!')

    def hole_punkte (self):
        '''
        Diese Methode gibt die Punkte der Funktionen des Dataframes zurürck.

        Input-Argument:
        -keine

        Output-Argument:
        - self.__dic_punkte = {'y1':[Punkt(x,y),Punkt(x,y),...],
                               'y2':[Punkt(x,y),...],...}
            Key = Funktionsname
            Value = Liste mit Instanzen der Klasse Punkt
        '''
        #Prüfen, ob dem Datensatz bereits ein Dataframe zugewiesen wurde
        if isinstance(self._dataframe,pd.DataFrame):
            #Erstellen des Dictionary
            self.__dic_punkte = {}
            #Iteration über die Spaltenbezeichnung des Dataframes (x,y1,..)
            for c in self.hole_spaltenbezeichnung():
                #Erstellen der Liste für die Punkte pro Spaltenbezeichnung
                self.__list_punkte = []
                #über die Spalte x soll nicht iteriert werden --> x-Werte für
                #alle Funktionen identisch
                if c == 'x':
                    continue

                #Iteration über die Y-Werte der jeweiligen Funktionen
                for i in range (self.anzahl_zeilen()):
                    #Erstellen einer Instanz Punkt und Hinzufügen zur Liste der Punkte
                    self.__list_punkte.append(Punkt(self._dataframe.loc[i,'x'],\
                                                    self._dataframe.loc[i,c]))
                #Komplette Liste eine Spaltenbezeichnung (Funktion) wird zum
                #Dictionary hinzugefügt
                self.__dic_punkte[c] = self.__list_punkte
            #Ausgabe des Dictionary
            return self.__dic_punkte

        else:
            raise DatensatzError('Dem Datensatz wurde kein Dataframe zugewiesen!')

    def export_dic (self):
```

5

```
            '''
            Diese Methode gibt den Dataframe als Dictionary zurück.

            Input-Argument:
            -keine

            Output-Argument:
            - self.__dataframe als Dictionary
            '''
            #Prüfen, ob dem Datensatz bereits ein Dataframe zugewiesen wurde
            try:
                if isinstance(self._dataframe,pd.DataFrame):
                    #raise DatensatzError('Dem Datensatz wurde kein Dataframe
                    #zugewiesen!')
                    return self._dataframe.to_dict('records')

                else:
                    #return self.__dataframe.to_dict('records')
                    raise DatensatzError

            except DatensatzError:
                print (DatensatzError().datensatz_message)

    class Punkt ():

        def __init__(self, x_wert,y_wert):
            '''
            Diese Methode wird beim Erstellen einer Instanz der Klasse Punkt ausgeführt.

            Input Argumente:
            - x_wert (Intger oder Float)
            - y_wert (Integer oder Float)

            Output Argumente:
            - keine
            '''
            #Prüfen, ob die übergebenen Werte vom Typ Float oder Integer sind
            if isinstance(x_wert, (float,int)) and isinstance(y_wert,(float,int)):
                #Zuordnung zu den lokalen Variablen
                self.__x_wert = x_wert
                self.__y_wert = y_wert
            #Fehlermeldung, wenn die Werte nicht vom Typ Intger oder Float sind
            else:
                raise PunktError (PunktError().punkt_message)

        def hole_x_wert (self):
            '''
            Diese Methode gibt den X-Wert zurück.

            Input Argumente:
            -keine

            Output Argument:
            - X-Wert
            '''
            #Fehler abfangen, falls dem Punkt kein X-Wert zugeordnet wurde.
            try:
                return self.__x_wert

            except AttributeError:
                print ('Dem Punkt ist kein x-Wert zugeordnet!')
```

6

```
def hole_y_wert (self):
    '''
    Diese Funktion wird beim gibt den Y-Wert zurück.

    Input Argumente:
    -keine

    Output Argument:
    - Y-Wert
    '''
    #Fehler abfangen, falls dem Punkt kein Y-Wert zugeordnet wurde.
    try:
        return self.__y_wert

    except AttributeError:
        print ('Dem Punkt ist kein y-Wert zugeordnet!')

def berechne_y_abweichung (self, p):
    '''
    Diese Methode berechnet die Y-Abweichung von zwei gegebenen Y-Werten.

    Input Argumente:
    -p (Instanz der Klasse Punkt)

    Output Argument:
    - y-Abweichung
    '''
    #Prüfen, ob p eine Instanz der Klasse Punkt ist.
    if isinstance(p,Punkt):
        #Berechnen der Y-Abweichung.
        self.__y_abweichung = self.__y_wert - p.hole_y_wert()

        return self.__y_abweichung
    #Ist p keine Instanz der Klasse Punkt --> PunktError
    else:
        raise PunktError(PunktError().punkt_instanz_message)

def setze_idealfunktion (self, idealfunktion):
    '''
    Diese Methode ermöglicht das Setzen einer Idealfunktion zu dem Punkt.

    Input Argumente:
    -idealfunktion  (Instanz der Klasse Idealfunktion)

    '''
    #Prüfen, ob idealfunktion eine Instanz der Klasse Idealfunktion ist.
    if isinstance(idealfunktion,Idealfunktion):
        self.__idealfunktion = idealfunktion

    #Wenn die Variable idealfunktion keine Instanz der Klasse Idealfunktion
    #ist --> Error
    else:
        raise PunktError('Die zugewiesene Variable ist keine Instanz der\
                          Klasse Idealfunktion!')

def hole_idealfunktion (self):
    '''
    Diese Methode gibt die dem Punkt zugewiesene Idealfunktion zurück.

    Input Argument:
```

7

```
            - keine

            Output Argument:
            - self.__idealfunktion (Instanz der Klasse Idealfunktion!)
            '''
            #Fehler abfangen, falls self.__idealfunkion nicht definiert ist
            try:
                return self.__idealfunktion

            except AttributeError:
                print ('Es wurde noch keine Idealfunktion bestimmt!')

        def setze_y_abweichung (self, y_abweichung):
            '''
            Diese Methode ermöglicht das zuweisen einer Y-Abweichung zu einem Punkt.

            Input Argument:
            - y_abweichung

            Output Argument:
            -keine
            '''
            self.__y_abweichung = y_abweichung

        def hole_y_abweichung (self):
            '''
            Diese Methode gibt die y_abweichung des Punktes zurück.

            Input Argument:
            - keine

            Output Argument:
            - y_abweichung
            '''
            #Abfangen des Fehlers, falls das Attribut self.__y_abweichung nicht
            #definiert ist.
            try:
                return self.__y_abweichung

            except AttributeError:
                print ('Es wurde noch keine Y-Abweichung berechnet!')

    class Funktion (Datensatz):

        def __init__ (self, funktion):
            '''
            Diese Methode wird beim Erstellen einer Instanz der Klasse Funktion
            ausgeführt.

            Input Argumente:
            - funktion (Instanz der Klasse Pandas Dataframe)
                - nur zwei Spalten (x,y)

            Output Argument:
            -keine

            '''
            #Abfrage, ob die Variable funktion eine Instanz der Klasse Pandas
            #DataFrame ist
            if isinstance(funktion, pd.DataFrame):
                #Zuweisung zu einer lokalen Variable
```

8

```
            Datensatz.__init__(self,funktion)

        else:
            raise FunktionError (FunktionError().funktion_message)

    def hole_funktionsname (self):
        '''
        Diese Methode gibt den Namen der Funktion zurück.

        Input Argumente:
        - keine

        Output Argument:
        - funktionsname (Bezeichnung der zweiten Spalte.)

        '''
        return self._dataframe.columns[1]

    def hole_y_werte (self):
        '''
        Diese Methode gibt die Y-Werte (zweite Spalte) der Funktion zurück.

        Input Argumente:
        - keine

        Output Argument:
        - Y-Werte

        '''
        return self._dataframe[self.hole_funktionsname()]

    def hole_funktion (self):
        '''
        Diese Methode gibt die Funktion (Pandas DataFrame) zurück.

        Input Argumente:
        - keine

        Output Argument:
        - funktion (Pandas Dataframe)

        '''
        return self._dataframe

class Trainingsfunktion (Funktion):
    '''
    Diese Klasse erbt von der Elternklasse Funktion. Sie dient dazu, um der
    Trainingsfunktion eine Idealfunktion zuweisen zu können.
    '''

    def __init__(self, funktion):
        '''
        Diese Methode wird beim Erstellen einer Instanz der Klasse
        Trainingsfunktion ausgeführt.
        '''
        #Init-Methode der Elternklasse
        Funktion.__init__(self, funktion)

    def suche_idealfunktion (self, datensatz):
        '''
        Diese Methode bestimmt durch Berechnungen eine Idealfunktion für die
```

9

```
Trainingsfunktion.

Input-Argument:
- datensatz (Instanz der Klasse Datensatz)

Output-Argumente:
-keine

'''
#Prüfen, ob datensatz eine Instanz der Klasse Datensatz ist.
if isinstance(datensatz, Datensatz):
    #Abspeichern von datensatz als lokales Attribut
    self.__datensatz_ideal = datensatz
    #self.__minimale Abweichung zum Bestimmen der minimalen Abweichung
    #--> Startwert muss groß sein
    self.__minimale_abweichung = 10000000.0
    #Iteration über die Spalten des Attributes self.__datensatz
    for c_i in self.__datensatz_ideal.hole_spaltenbezeichnung():
        #Spaltenbezeichnung x soll ignoriert werden, da sie 'fix' ist.
        if c_i == 'x':
            #Nächster Schleifendurchlauf
            continue
        #Erstellen des lokalen Attributes self.__vergleichsfunktion
        #(Instanz der Klasse Funktion)
        self.__vergleichsfunktion = Funktion(pd.concat(\
            [self.__datensatz_ideal.hole_x_spalte(),\
             self.__datensatz_ideal.hole_y_spalte(c_i)],axis=1))
        #Bestimmen der maximalen Y-Abweichung (für Aufgabenteil 2)
        self.__maximale_y_abweichung = 0

        self.__sum_leastsquare = 0
        #Liste der Y-Abweichungen (für späteren Plot)
        self.__list_y_abw = []
        #Iteration über die Zeilen der Spalte c_i
        for i in range (self.anzahl_zeilen()):
            #Y-Wert der Trainingsfunktion - Y-Wert der Vergleichsfunktion
            self.__y_abweichung = self._dataframe.loc[i,\
                self.hole_funktionsname()]-\
                self.__vergleichsfunktion.hole_funktion().loc[i,\
                    self.__vergleichsfunktion.hole_funktionsname()]
            #Hinzufügen der Y-Abweichung zur Liste
            self.__list_y_abw.append(self.__y_abweichung)
            self.__sum_leastsquare += (self.__y_abweichung)**2
            #Wenn die Y-Abweichung größer als self.__maximale_y_abweichung
            #--> neue mximale Y-Abweichung
            if self.__y_abweichung > self.__maximale_y_abweichung:
                self.__maximale_y_abweichung = self.__y_abweichung

        #Finale Berechnung der Least-Square-Funktion
        self.__sum_leastsquare = self.__sum_leastsquare/self.anzahl_zeilen()


        #Wenn Least Square größer als minimale abweichung --> neue
        #minimale Abweichung und neue Idelafunktion
        if self.__sum_leastsquare < self.__minimale_abweichung:
            self.__minimale_abweichung = self.__sum_leastsquare
            #Neue Zuweisung für das Attribut self.__ideal_funktion
            #(Instanz der Klasse Idealfunktion)
            self.__ideal_funktion= Idealfunktion (\
                self.__vergleichsfunktion.hole_funktion(),\
                self._dataframe, self.__sum_leastsquare, \
```

10

```
                        self.__maximale_y_abweichung,self.__list_y_abw)
            else:

                raise DatensatzError (DatensatzError().datensatz_funktion_message)

        def setze_idealfunktion (self, idealfunktion):
            '''
            Diese Methode ermöglicht das Zuweisen einer Idealfunktion zur
            Trainingsfunktion ohne Berechnung.

            Input Argumente:
            -idealfunktion (Instanz der Klasse Idealfunktion)

            Output Argument:
            -keine
            '''
            if isinstance(idealfunktion, Idealfunktion):
                self.__ideal_funktion = idealfunktion

            else:
                raise FunktionError('Die Variable idealfunktion ist keine Instanz\
                                    der Klasse Idealfunktion.')

        def hole_idealfunktion (self):
            '''
            Diese Methode gibt die Idealfunktion der Trainingsfunktion zurück.

            Input Argumente:
            -keine

            Output Argument:
            -self.__ideal_funktion (Instanz der Klasse Idealfunktion.)
            '''
            #Abfangen des Fehler, falls das Attribut self.__ideal_funktion nicht
            #definiert ist.
            try:
                return self.__ideal_funktion

            except AttributeError:
                print ('Es wurde noch keine Idealfunktion bestimmt!')

    class Idealfunktion (Funktion, Datensatz):

        def __init__(self, funktion, trainingsfunktion, least_square,\
                     maximale_y_abweichung, list_y_abweichungen):
            '''
            Diese Methode wird beim Erstellen einer Instanz der Klasse Idealfunktion
            ausgeführt.

            Input Argumente:
            - funktion (Instanz der Klasse Pandas Dataframe)
                - nur zwei Spalten (x,y)
            - trainingsfunktion (Instanz der Klasse Trainingsfunktion)
            - least-square (Least-Square-Abweichung von Idealfunktion zur
                            Trainingsfunktion)
            - maximale_y_abweichung (Maximale Y-Abweichung zwischen den Y-Werten)
            - list_y_abweichung (Liste aller Y-Abweichungen)

            Output Argument:
            -keine
            '''
```

11

```
        #Ausführen der Init-Methode der Elternklasse
        Funktion.__init__(self, funktion)
        #Zuweisen zu lokalen Attributen
        self.__train_funktion = trainingsfunktion
        self.__least_square = least_square
        self.__maximale_y_abweichung = maximale_y_abweichung
        self.__list_y_abweichungen = list_y_abweichungen

    def zuweisung_testpunkte (self, testdatensatz, testkriterium):
        #Festlegen des Testkriterium, hier Faktor Wurzel 2

        if isinstance(testdatensatz,Datensatz):
            #Punkte der Idealfunktionen und Testdaten extrahieren
            punkte_test = testdatensatz.hole_punkte()

            #Festlegen der Struktur eines Dictionaries für Aufgabenteil 2
            #--> Testpunkte zur Idealfunktion
            p_test_data = {'X (Testfunktion)':[],'Y (Testfunktion)':[],\
                           'Delta Y':[],'Idealfunktion':[]}
            df_p_test = pd.DataFrame(p_test_data)

            #Iteration über die Values des Dictionaries der Punkte Test
            #--> Liste mit Instanzen der Klasse Punkt
            for v in punkte_test.values():
                #Iteration über die Instanzen der Klasse Punkt
                for p_test in v:
                    #Es wird für jeden Punkt aus dem Testdatensatz überprüft,
                    #ob er zu einer Idealfunktion passt.
                    #Iteration über das Dictionary der Idealfunktionen
                    for p_ideal in self.hole_punkte()[self.hole_funktionsname()]:
                        #Prüfen, ob beide Punkte den gleichen X-Wert habe
                        if p_test.hole_x_wert() == p_ideal.hole_x_wert():
                            #Berechnung der Y-Abweichung und Multiplikation mit
                            #dem Testkriterium
                            y_abweichung = p_test.berechne_y_abweichung(p_ideal)
                            y_abw_testkriterium = y_abweichung * testkriterium
                            '''Prüfen, ob die Abweichung inkl. Faktor des
                            Testkriteriums kleiner als die maximale Y-Abweichung
                            von der Idealfunktion zur Trainingsfunktion ist'''
                            if abs(y_abw_testkriterium) <= \
                                abs(self.hole_maximale_y_abweichung()):
                                #Füllen des Panda Dataframes für Aufgabenteil 2
                                df_temp_data = {'X (Testfunktion)':\
                                                [p_test.hole_x_wert()],\
                                                'Y (Testfunktion)':\
                                                [p_test.hole_y_wert()],\
                                                'Delta Y':[y_abweichung],\
                                                'Idealfunktion':\
                                                [self.hole_funktionsname()]}
                                df_temp = pd.DataFrame(df_temp_data)
                                df_p_test = pd.concat([df_p_test,df_temp])

                            else:
                                continue
                        else:
                            continue
            return df_p_test

        else:
            raise DatensatzError('Die Variable testdatensatz ist keine Instanz\
                                  der Klasse Datensatz.')
```

12

```
def setze_trainingsfunktion (self,trainingsfunktion):
    '''
    Diese Methode erlaubt das Zuweisen einer Trainingsfunktion

    Input Argumente:
    -trainingsfunktion (Instanz der Klasse Trainingsfunktion)

    Output Argument:
    -keine
    '''
    #Prüfen, ob trainingsfunktion eine Instanz der Klasse Trainingsfunktion ist.
    if isinstance(trainingsfunktion, Trainingsfunktion):
        self.__train_funktion = trainingsfunktion

    else:
        raise FunktionError('Die Variable trainingsfunktion ist keine\
                             Instanz der Klasse Trainingsfunktion.')

def hole_trainingsfunktion (self):
    '''
    Diese Methode gibt die Trainingsfunktion zurück.

    Input Argumente:
    -keine

    Output Argument:
    -self.__train_funktion (Instanz der Klasse Trainingsfunktion)
    '''
    return self.__train_funktion

def setze_least_square (self, least_square):
    '''
    Diese Methode erlaubt das Zuweisen eines Least Square Wertes.

    Input Argumente:
    -least_square

    Output Argument:
    -keine
    '''
    #Prüfen, ob least_square eine Zahl ist.
    if isinstance(least_square, (float,int)):
        #Zuweisen zu einer lokalen Variable
        self.__least_square = least_square

    else:
        raise FunktionError('Der zugewiesene Least-Square-Wert ist keine\
                             Zahl!')

def hole_least_square (self):
    '''
    Diese Methode gibt den Least-Square-Wert zurück.

    Input Argumente:
    -keine

    Output Argument:
    -self.__least_square
    '''
    return self.__least_square
```

13

```
def setze_maximale_y_abweichung (self, maximale_y_abweichung):
    '''
    Diese Methode erlaubt das Zuweisen eines Maximale-Y-Abweichung Wertes.

    Input Argumente:
    -maximale_y_abweichung

    Output Argument:
    -keine
    '''
    #Prüfen, ob maximale_y_abweichung eine Zahl ist.
    if isinstance(maximale_y_abweichung, (float,int)):
        #Zuweisen zu einer lokalen Variable
        self.__maximale_y_abweichung = maximale_y_abweichung

    else:
        raise FunktionError('Der zugewiesene Maximale-Y-Abweichung-Wert ist\
                             keine Zahl!')

def hole_maximale_y_abweichung (self):
    '''
    Diese Methode gibt den Maximale-Y-Abweichung-Wert zurück.

    Input Argumente:
    -keine

    Output Argument:
    -self.__maximale_y_abweichung
    '''
    return self.__maximale_y_abweichung

def setze_y_abweichung (self,list_y_abw):
    '''
    Diese Methode erlaubt das Zuweisen einer Liste mit Y-Abweichungen.

    Input Argumente:
    -list_y_abw (Liste)

    Output Argument:
    -keine
    '''
    #Prüfen, ob list_y_abw eine Instanz der Klasse Liste ist.
    if isinstance(list_y_abw, list):
        #Zuweisen zu einer lokalen Variable
        self.__list_y_abweichungen = list_y_abw

    else:
        raise FunktionError('Der zugewiesene Y-Abweichungen sind keine\
                             Liste!')

def hole_y_abweichungen (self):
    '''
    Diese Methode gibt die Y-Abweichungen zurück.

    Input Argumente:
    -keine

    Output Argument:
    -self.__list_y_abweichungen (Liste)
    '''
```

```
        return self.__list_y_abweichungen


#Trainingsdatensatz erstellen (Instanz der Klasse Datensatz)
trainingsdatensatz = Datensatz()
trainingsdatensatz.import_dataframe('Beispiel_Datensaetze/train.csv')

#Idealdatensatz erstellen (Instanz der Klasse Datensatz)
idealdatensatz = Datensatz()
idealdatensatz.import_dataframe('Beispiel_Datensaetze/ideal.csv')

#Testdatensatz erstellen (Instanz der Klasse Datensatz)
testdatensatz = Datensatz()
testdatensatz.import_dataframe('Beispiel_Datensaetze/test.csv')

#Erstellen zweier Dictionaries zur Verwaltung der Trainings- und Idealfunktionen
dic_trainingsfunktionen = {}
dic_idealfunktionen = {}

'''
Füllen der Dictionaries und Zuweisen der Idealfunktionen zur einer
Trainingsfunktion.
Struktur der Dictionaries :{Funktionsname (z.B. y1):Instanz der Klasse
                            Trainings-oder Idealfunktion,y2:....}
'''
for d in trainingsdatensatz.hole_spaltenbezeichnung():
    if d == 'x':
        continue
    trainingsfunktion = Trainingsfunktion(pd.concat(\
                            [trainingsdatensatz.hole_x_spalte(),\
                             trainingsdatensatz.hole_y_spalte(d)],axis=1))

    trainingsfunktion.suche_idealfunktion(idealdatensatz)
    dic_trainingsfunktionen [trainingsfunktion.hole_funktionsname()] =\
        trainingsfunktion
    dic_idealfunktionen [trainingsfunktion.hole_idealfunktion().hole_funktionsname()]=\
        trainingsfunktion.hole_idealfunktion()


x_spalte = False
#Iteration über das Dicitionary mit den Idealfunktionen
#(Struktur:{Funktionsname:Funktion,Funktionsname:Funktion,...})
#Ziel: Erstellen einer Instanz der Klasse Datensatz für die vier Idealfunktionen
for key,value in dic_idealfunktionen.items():
    #key = Funktionsname, value=Instanz der Klasse Idealfunktion

    if x_spalte == False:
        #Erstellen eines Datensatzes mit dem ersten Eintrag des Dictionarys
        #(zwei Spalten, x und y)
        idealfunktionen_df = Datensatz (value.hole_funktion())
        x_spalte = True
    else:
        #Datensatz idealfunktionen_df erweiterten um die Y-Spalte des nächsten
        #Eintrages des Dictionarys
        idealfunktionen_df = Datensatz (pd.concat([\
                            idealfunktionen_df.hole_dataframe(),\
                                value.hole_y_werte()], axis =1))

#Festlegen des Testkriterium, hier Faktor Wurzel 2
testkriterium = math.sqrt(2)
```

15

```
#Punkte der Idealfunktionen und Testdaten extrahieren
punkte_test = testdatensatz.hole_punkte()

#Festlegen der Struktur eines Dictionaries für Aufgabenteil 2 --> Testpunkte
#zur Idealfunktion
p_test_data = {'X (Testfunktion)':[],'Y (Testfunktion)':[],\
               'Delta Y':[],'Idealfunktion':[]}

df_p_testdaten = pd.DataFrame(p_test_data)

for ideal_f in dic_idealfunktionen.values():
    #ds_temp = ideal_f.zuweisung_testpunkte(testdatensatz,testkriterium)
    df_temp = ideal_f.zuweisung_testpunkte(testdatensatz,testkriterium)

    df_p_testdaten = pd.concat([df_p_testdaten,df_temp])

#Erstellen einer Liste mit den Indizes für den DataFrame df_pp_test
liste_index=[]
for index in range (df_p_testdaten.shape[0]):
    liste_index.append(index)
#Zuweisen der Indizes
df_p_testdaten.index=liste_index

#Erstellen einer Instanz der Klasse Datensatz
p_test_anpassbar = Datensatz(df_p_testdaten)

style.use('ggplot')

#Plotten der Ergebnisse für Aufgabenteil 1
for key,value in dic_trainingsfunktionen.items():
    '''
    ax1 = Plot der Trainingsfunktion
    ax2 = Plot der Idealfunktion
    ax3 = Plot der Y-Abweichungen
    '''
    fig, (ax1,ax2,ax3) = plt.subplots(nrows=3, ncols=1, figsize=(20,20))

    ax1.plot(dic_trainingsfunktionen[key].hole_x_spalte(),\
             dic_trainingsfunktionen[key].hole_y_werte(),\
             label = 'Trainingsfunktion '+key)
    ax2.plot(dic_trainingsfunktionen[key].hole_idealfunktion().hole_x_spalte(),\
             dic_trainingsfunktionen[key].hole_idealfunktion().hole_y_werte(),\
                 label = 'Idealfunktion '+ \
                 dic_trainingsfunktionen[key].hole_idealfunktion().hole_funktionsname())
    ax3.plot(dic_trainingsfunktionen[key].hole_idealfunktion().hole_x_spalte(),\
             dic_trainingsfunktionen[key].hole_idealfunktion().hole_y_abweichungen(),\
                 label = 'Abweichungen (y)')
    ax1.set_xlabel("X-Werte", fontsize=14)
    ax1.set_ylabel("Y-Werte", fontsize=14)
    ax2.set_xlabel("X-Werte", fontsize=14)
    ax2.set_ylabel("Y-Werte", fontsize=14)
    ax3.set_xlabel("X-Werte", fontsize=14)
    ax3.set_ylabel("Y-Werte", fontsize=14)
    ax1.legend(fontsize ="15", loc = "lower right")
    ax2.legend(fontsize ="15", loc = "lower right")
    ax3.legend(fontsize ="15", loc = "lower right")
    ax1.set_title('Trainingsfunktion'+' '+key, fontsize = 40)

    fig.savefig('Trainingsfunktion'+key+'.png')

#Plotten der Idealfunktionen inkl. passender Punkte
```

16

```
#Iteration über das Dicitionary der Idealfunktionen
for key, value in dic_idealfunktionen.items():
    #key = Funktionsname, value = Instanz der Klasse Idealfunktion
    #Erzeuge von leeren Listen (X bzw Y-Liste)
    pp_liste_x = []
    pp_liste_y = []
    #Iteration über die Instanz der Klasse Datensatz
    for i in range (p_test_anpassbar.anzahl_zeilen()):
        #Abfrage ob Funktionsnamen identisch sind

        if p_test_anpassbar.hole_dataframe().loc[i,'Idealfunktion'] == key:
            pp_liste_x.append(p_test_anpassbar.hole_dataframe().iloc[i,0])
            pp_liste_y.append(p_test_anpassbar.hole_dataframe().iloc[i,1])

        else:
            continue
    '''
    ax.plot = Plot der Idealfunktion
    ax.scatter = Plot der passenden Punkte
    '''
    fig, ax = plt.subplots(figsize=(20,20))
    ax.plot(dic_idealfunktionen[key].hole_x_spalte(),\
            dic_idealfunktionen[key].hole_y_werte(), \
                label = 'Idealfunktion '+key )
    ax.scatter(pp_liste_x,pp_liste_y,label = "Testpunkte", color ="blue")
    ax.set_xlabel("X-Werte", fontsize=14)
    ax.set_ylabel("Y-Werte", fontsize=14)
    ax.legend(fontsize = "15", loc="lower right")
    plt.title ('Idealfunktion mit passenden Testpunkten', fontsize ="40")
    fig.savefig ('Idealfuntionfunktion_'+key+'_Testpunkte.png')

#Liste der Datensätze
datensaetze_list = [trainingsdatensatz, idealdatensatz, p_test_anpassbar]
#Liste der Dateinamen für die SQLite-Datenbank (gleiche Reihenfolge wie
#Liste der Datensätze)
dateinamen_list = ['Trainingsfunktion', 'Idealfunktionen', 'Testdaten']

#Für jeden Datensatz wird eine separate SQLite-Datei angelegt
for dn in range (len(dateinamen_list)):
    #Herstellen der Verbindung zur Datenbank -Speicherort im selben Ordner
    engine = create_engine('sqlite+pysqlite:///'+dateinamen_list[dn]+'.sqlite',\
                           echo=True)
    conn= engine.connect()
    #Falls die SQLite-Datei bereits existiert und Tabellen enthält --> Löschen
    conn.execute(text('DROP TABLE IF EXISTS '+ dateinamen_list[dn]))
    #Exportieren des DataFrames in die SQLlite-Datei
    datensaetze_list[dn].hole_dataframe().to_sql(dateinamen_list[dn],\
                                                 conn, if_exists='replace',\
                                                 index=False)
    conn.commit()
    #Trennen der Verbindung zur SQLite-Datenbank
    conn.close()
```

17

A.2 Python-Programm (UnitTest)

```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr  6 13:47:13 2024

@author: bauma
"""
import unittest
import Hausarbeit as ha
import pandas as pd

class UnitTestHausarbeit(unittest.TestCase):

    def test_init_datensatz(self):
        '''
        Diese Methode prüft, ob ein DatensatzError ausgegeben wird, wenn der
        Instanz der Klasse Datensatz eine nichtleere Zeichenkette übergeben wird.
        '''
        with self.assertRaises(ha.DatensatzError):
            ds = ha.Datensatz('Test')

    def test_init_funktion(self):
        '''
        Diese Methode prüft, ob ein FunktionError ausgegeben wird, wenn der
        Instanz der Klasse Funktion eine nichtleere Zeichenkette übergeben wird.
        '''
        with self.assertRaises(ha.FunktionError):
            f = ha.Funktion('Test')

    def test_init_punkt(self):
        '''
        Diese Methode prüft, ob ein PunktError ausgegeben wird, wenn der
        Instanz der Klasse Punkt zwei Zeichenketten übergeben wird.
        '''
        with self.assertRaises(ha.PunktError):
            p = ha.Punkt('Test','Test2')

    def test_p_hole_x_wert(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Punkt den richtigen Wert
        zurück gibt, wenn der x-Wert ausgegeben werden soll.
        '''
        d_x = ha.Punkt (2,3)
        result = d_x.hole_x_wert()
        self.assertEqual(result,2,"Der X-Wert sollte 2 ergeben")


    def test_p_hole_y_wert(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Punkt den richtigen Wert
        zurück gibt, wenn der y-Wert ausgegeben werden soll.
        '''
        d_y = ha.Punkt (2,3)
        result = d_y.hole_y_wert()
        self.assertEqual(result,3,"Der Y-Wert sollte 3 ergeben")

    def test_pp_berechne_y_abweichung(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Punkt die richtige
        Y-Abweichung berechnet.
        '''
```

1

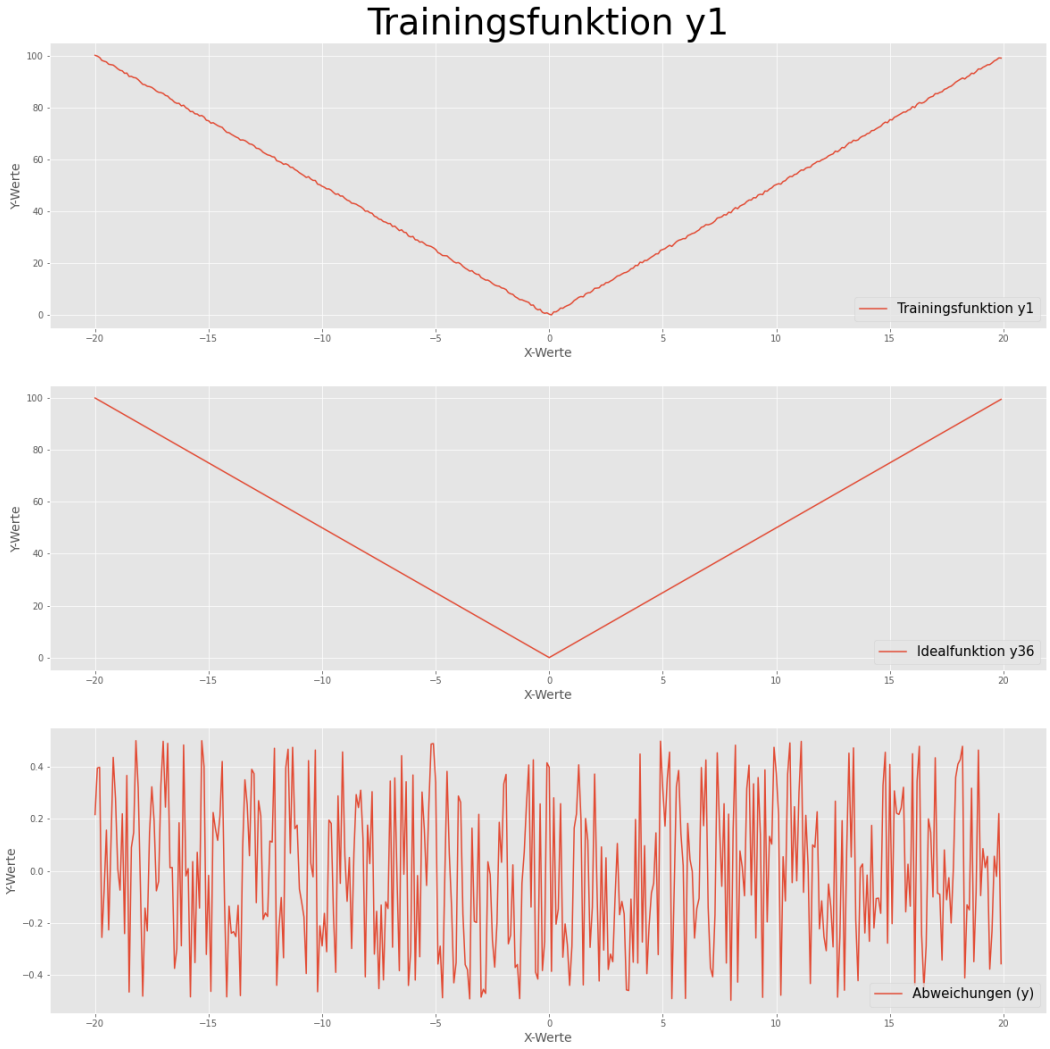
```
        p_1 = ha.Punkt(2,3)
        p_2 = ha.Punkt(1,2)
        result = p_1.berechne_y_abweichung(p_2)
        self.assertEqual(result,1,"Die Y - Abweichung sollte 1 betragen")

    def test_hole_y_abweichung(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Punkt die richtige
        Y-Abweichung zurück gibt.
        '''
        d_y = ha.Punkt (2,3)
        d_y.setze_y_abweichung(2)
        result = d_y.hole_y_abweichung()
        self.assertEqual(result, 2, 'Die Y-Abweichung sollte 2 sein!')

    def test_hole_y_spalte(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Funktion die richtigen
        Y-Werte zurückgibt.
        '''
        df_dic = {"x":[1,2,3],"Y1":[4,5,6]}
        df = pd.DataFrame(df_dic)
        f_y = ha.Funktion (df)
        result = []
        for i in f_y.hole_y_spalte('Y1'):
            result.append(i)
        self.assertEqual(result, [4,5,6], 'Die Y-Werte sollten 4,5,6 sein')

    def test_hole_x_spalte(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Funktion die richtigen
        X-Werte zurückgibt.
        '''
        df_dic = {"x":[1,2,3],"Y1":[4,5,6]}
        df = pd.DataFrame(df_dic)
        f_x = ha.Funktion (df)
        result = []
        for i in f_x.hole_x_spalte():
            result.append(i)
        self.assertEqual(result, [1,2,3], 'Die X-Werte sollten 1,2,3 sein')

    def test_hole_anzahl_zeilen(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Funktion die richtige
        Anzahl an Werten zurückgibt.
        '''
        df_dic = {"x":[1,2,3],"Y1":[4,5,6]}
        df = pd.DataFrame(df_dic)
        f_w = ha.Funktion (df)
        result = f_w.anzahl_zeilen()

        self.assertEqual(result,3, 'Die Anzahl der Werte sollte 3 betragen')

    def test_hole_funktionsname(self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Funktion den richtigen
        Funktionsnamen zurückgibt
        '''
        df_dic = {"X":[1,2,3],"Y1":[4,5,6]}
        df = pd.DataFrame(df_dic)
```

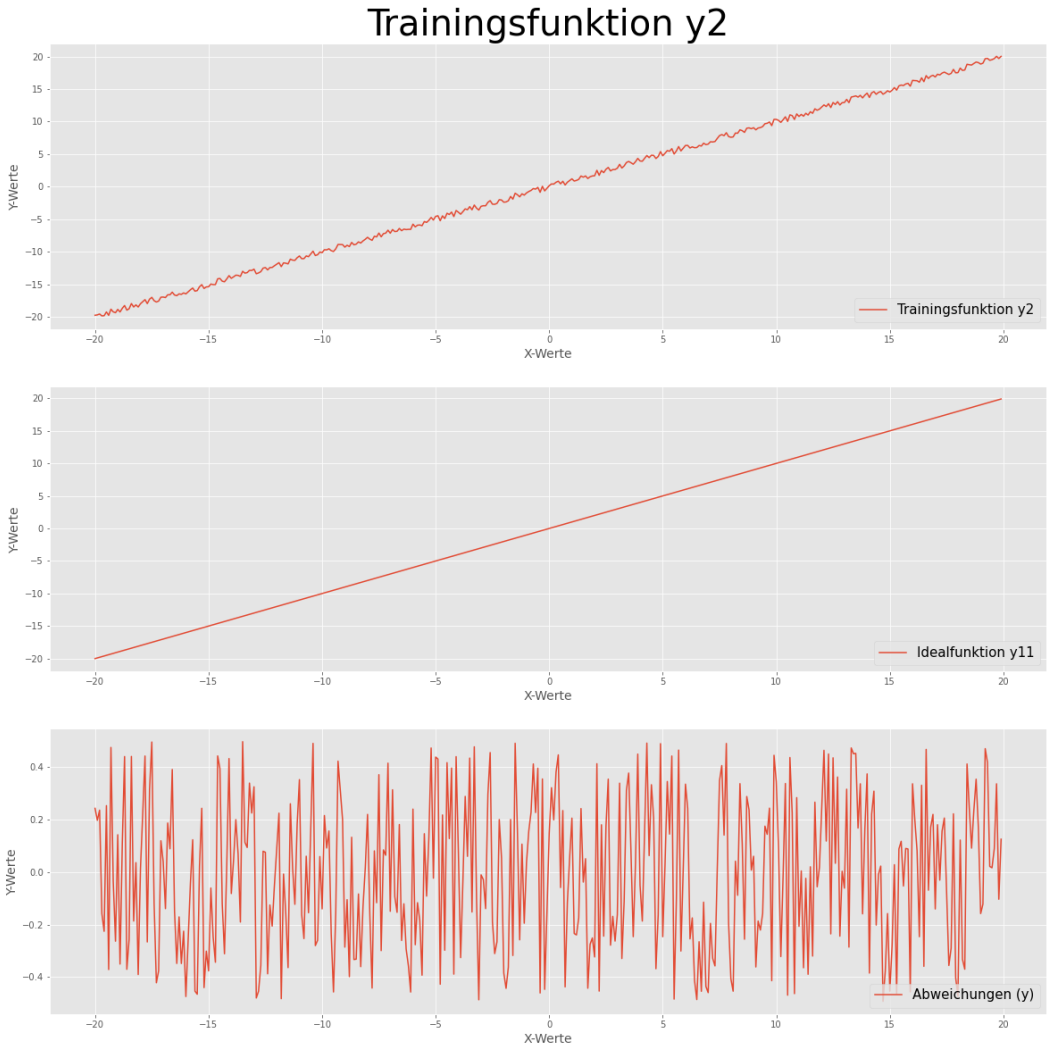
2

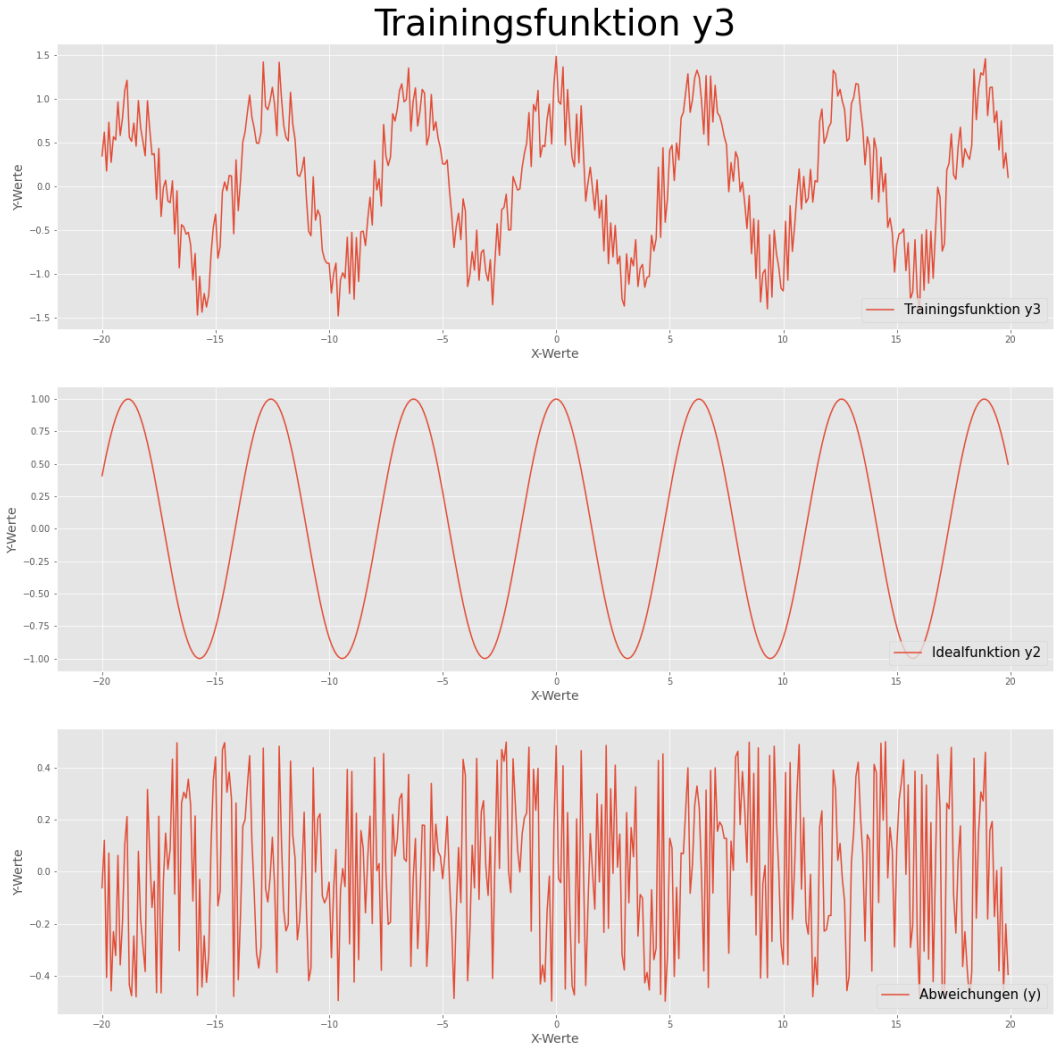
```
        f_f = ha.Funktion (df)
        result = f_f.hole_funktionsname()
        self.assertEqual(result, 'Y1', 'Der Funktionsname sollte Y1 sein')

    def test_suche_idealfunktion (self):
        '''
        Diese Methode prüft, ob eine Instanz der Klasse Funktion die richtige
        Idealfunktion bestimmt
        '''
        df_dic = {"x":[1,2,3],"y1":[4,5,6]}
        df = pd.DataFrame(df_dic)
        f_i = ha.Trainingsfunktion (df)
        df_dic_ds = {"x":[1,2,3],"y1":[4.2,5.2,6.2],"y2":[5,6,7]}
        df_ds = pd.DataFrame(df_dic_ds)
        ds = ha.Datensatz(df_ds)
        f_i.suche_idealfunktion(ds)
        result = f_i.hole_idealfunktion().hole_funktionsname()

        self.assertEqual(result,'y1' , 'Der Funktionsname sollte Y1 sein')

if __name__ == '__main__':
    unittest.main()
```

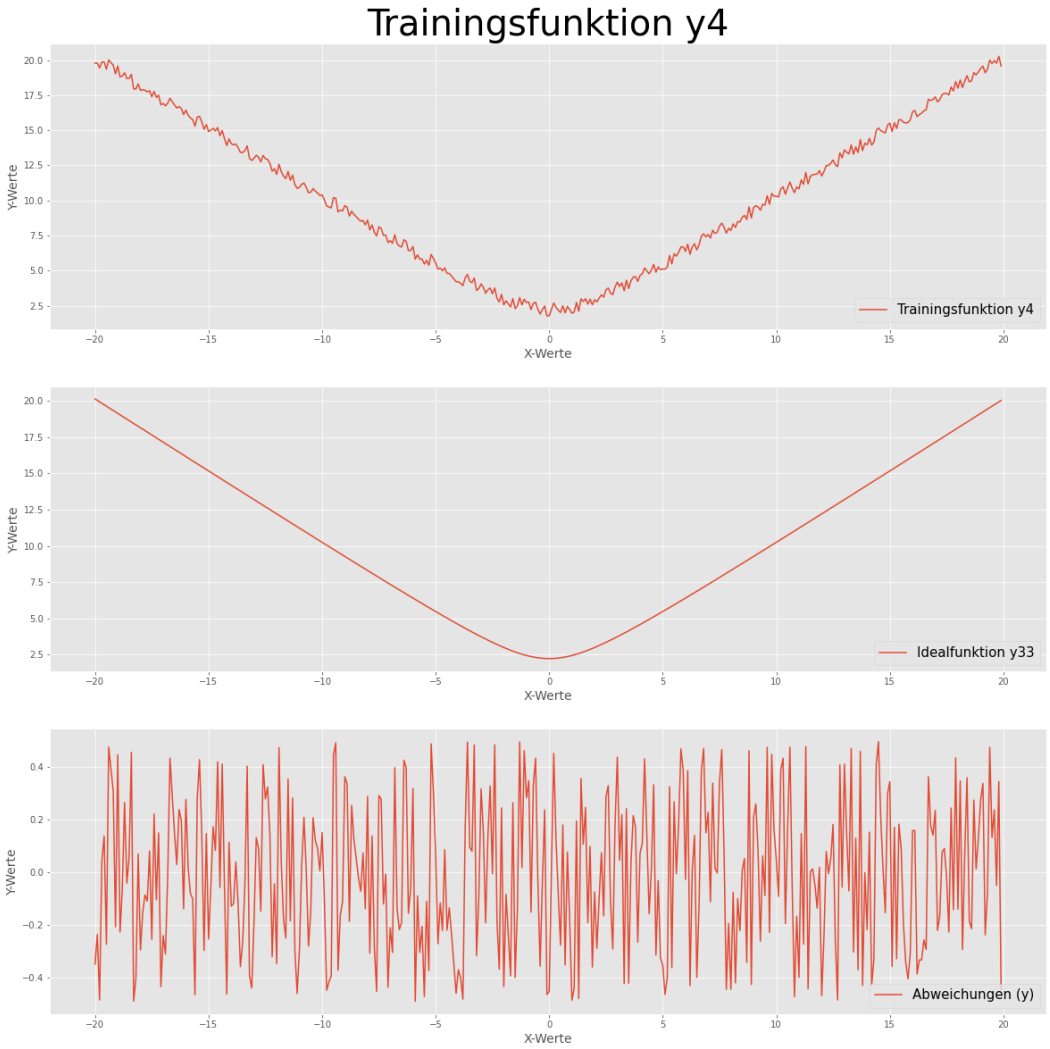
3

A.3 Visualisierungen Aufgabenteil 1

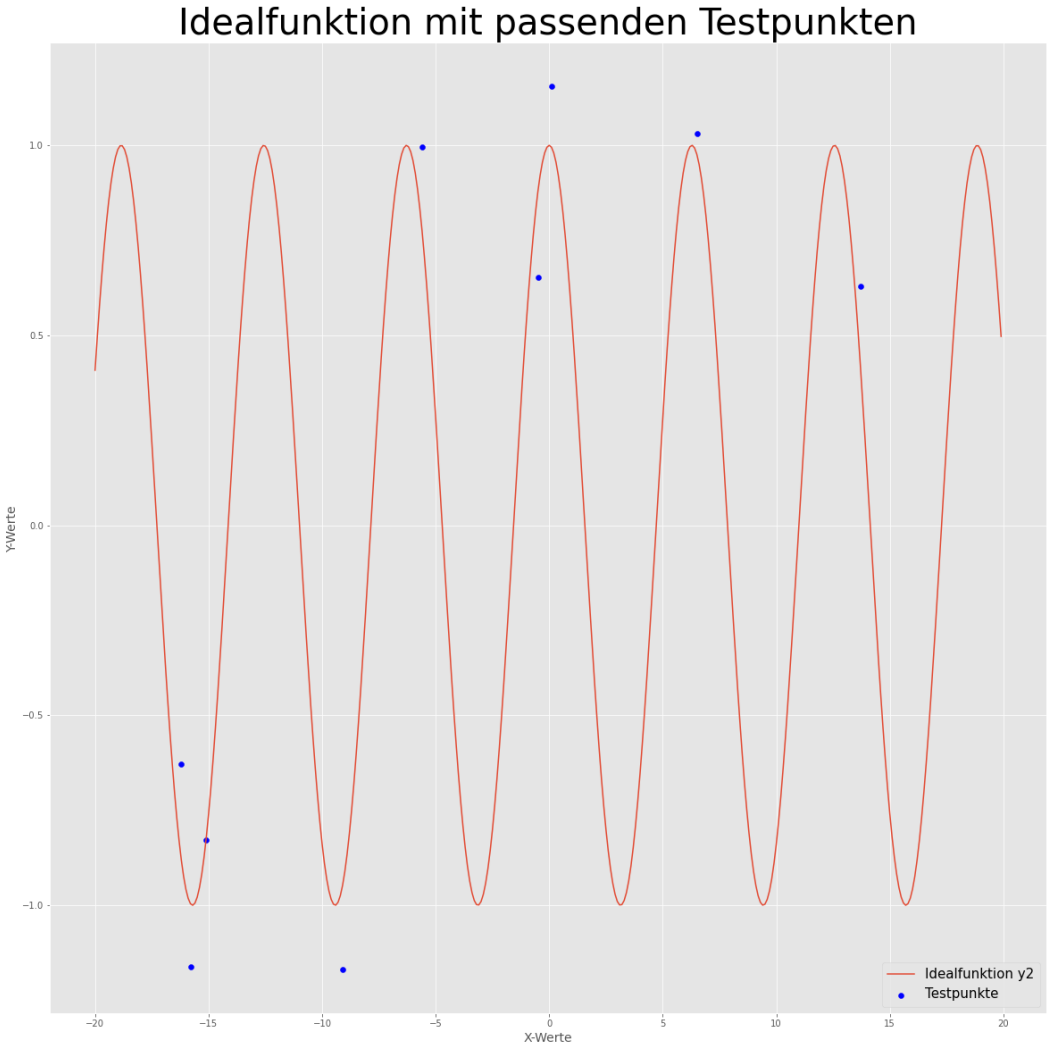


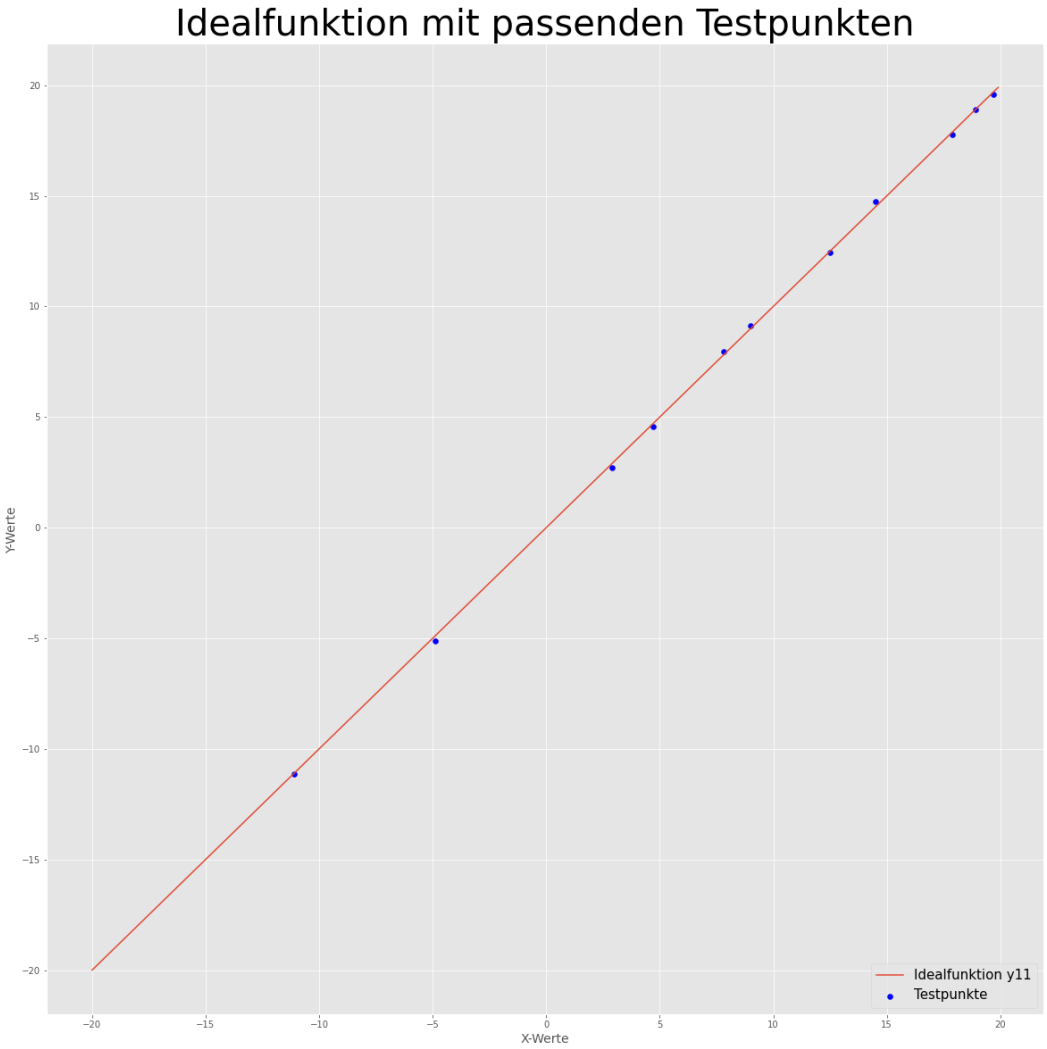

Trainingsfunktion y2
Y-Werte
X-Werte
Trainingsfunktion y2
Y-Werte
X-Werte
Idealfunktion y11
Y-Werte
X-Werte
Abweichungen (y)

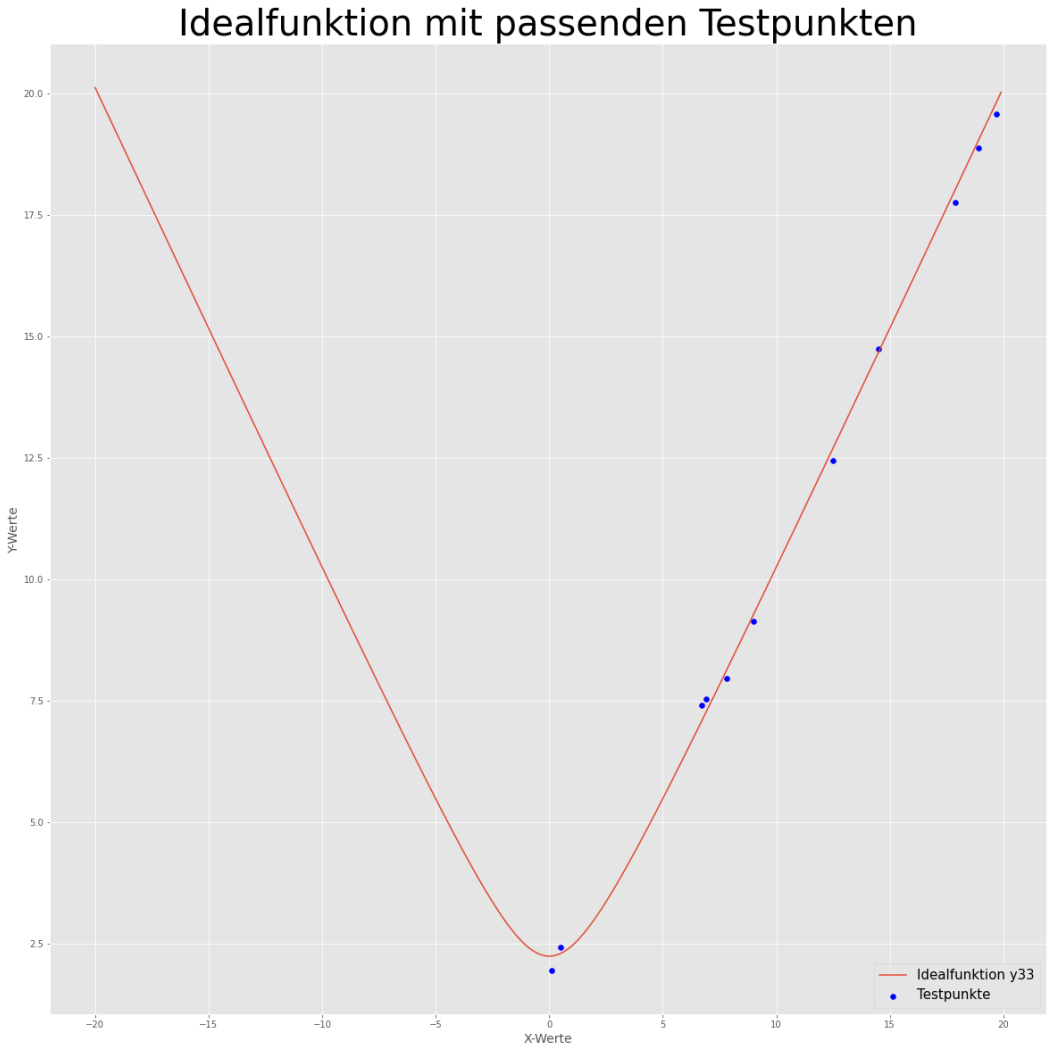

Trainingsfunktion y3
Y-Werte
X-Werte
Trainingsfunktion y3
Idealfunktion y2
Abweichungen (y)

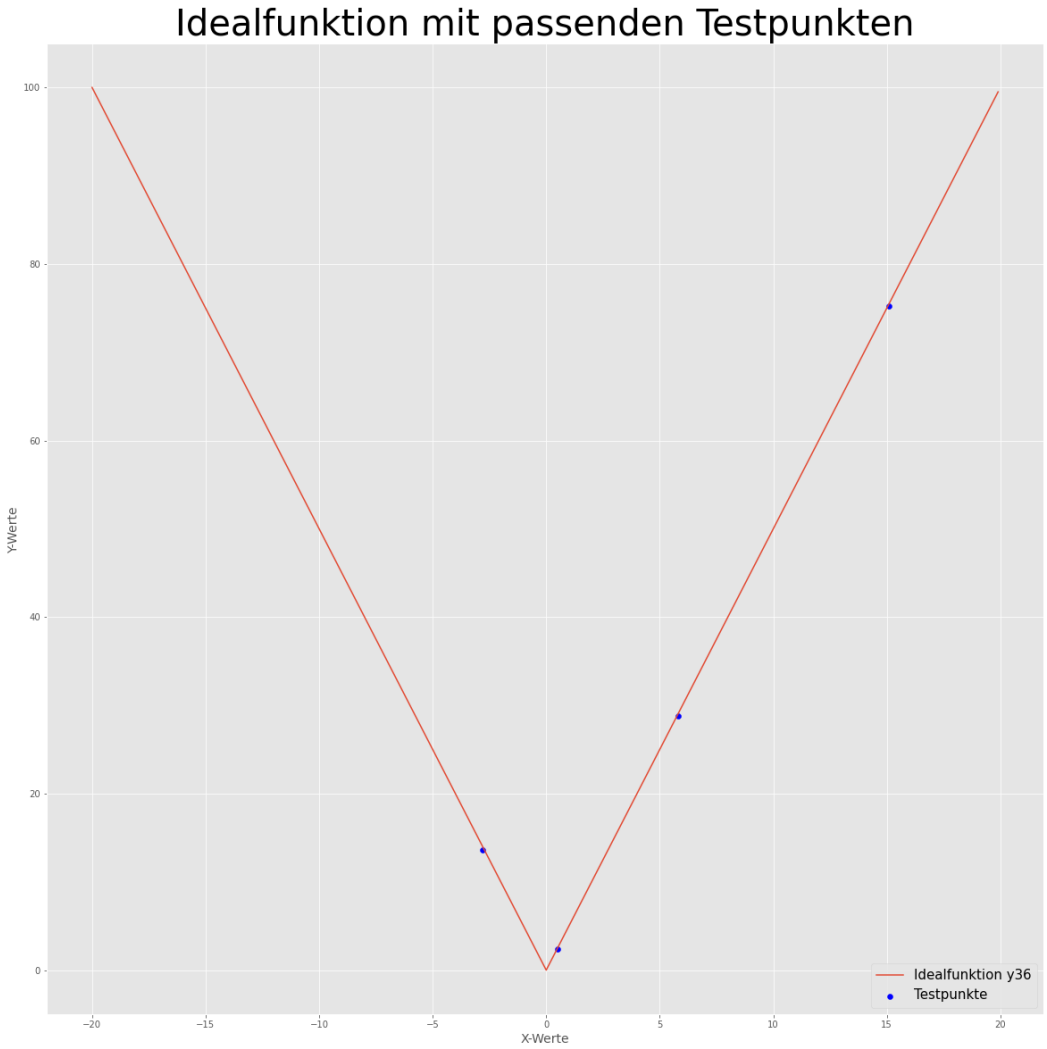

Trainingsfunktion y4
Y-Werte
X-Werte
Trainingsfunktion y4
Idealfunktion y33
Abweichungen (y)

A.4 Visualisierungen Aufgabenteil 2




Idealfunktion mit passenden Testpunkten
Y-Werte
X-Werte
Idealfunktion y11
Testpunkte


Idealfunktion mit passenden Testpunkten
Y-Werte
X-Werte
Idealfunktion y33
Testpunkte


Idealfunktion mit passenden Testpunkten
Y-Werte
X-Werte
Idealfunktion y36
Testpunkte