FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

FH Aachen

Fachbereich
Elektrotechnik und Informationstechnik

Bachelorarbeit

Prognose der Anwesenheit von Personen für die Gebäudeautomatisierung mittels Umweltsensordaten

Alexander Loosen
Matr.-Nr.: 3167353

Referent: Prof. Dr-Ing. Ingo Elsen

6. April 2022

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, 6. April 2022

Geheimhaltung - Sperrvermerk

Die vorliegende Arbeit unterliegt bis [Datum] der Geheimhaltung. Sie darf vorher weder vollständig noch auszugsweise ohne schriftliche Zustimmung des Autors, des betreuenden Referenten bzw. der Firma [Firmenname und -sitz] vervielfältigt, veröffentlicht oder Dritten zugänglich gemacht werden.

Inhalt

1. Einleitung

Gebäudeautomatisierung bezeichnet die automatische Steuerung und Regelung von Gebäudetechnik wie Heizung, Lüftung oder Beleuchtung. Während sie bisher hauptsächlich für die Optimierung der Energieeffizienz von gewerblichen und öffentlichen Gebäuden genutzt wurde, welche in Zuge solcher Optimierungsschritte als „Smart Buildings" bezeichnet werden, rückt sie in den letzten Jahren zunehmend unter dem Begriff „Smart Home" auch in den privaten Bereich.
Die beiden Begriffe stehen in den letzten Jahren so im Vordergrund, weil eine Verbesserung der Energieeffizienz durch bauphysikalische Maßnahmen, wie verminderte Wärmeverluste durch bessere Isolation, an ihre Grenzen gestoßen sind.

In Deutschland haben private Haushalte einen Anteil am gesamten Energieverbrauch des Landes von etwa 29%[1]. Davon nimmt die Erzeugung der Raumwärme einen Anteil von etwa 68%[2] ein. Vorallem bezogen auf den Energieverbrauch durch das Heizen stellt die Gebäudeautomatisierung also einen der größten Sektoren dar, in denen eine Optimierung der Prozesse entscheidend wäre. Beispielsweise können Wärmeverluste über die Nacht hinweg um 20% verringert werden, wenn Abends die Rolläden heruntergelassen werden.

Zur weiteren Steigerung der Energieeffizienz ist es also nötig, die Gebäudetechnik insofern automatisch anzusteuern, sodass sog. Performance-Gaps vermieden werden. Performance-Gaps stellen eine Diskrepanz im Energieverbrauch eines Gebäudes zwischen einem theoretischen Soll-Wert zu einem tatsächlichem Ist-Wert dar.

1.1. Motivation und Aufgabenstellung

Für nahezu alle Bereiche der Gebäudeautomatisierung stellt die Anwesenheit von Personen eine zentrale Variable dar. Es ist entscheidend bei einem Optimierungsprozess zu erkennen, zu welchen Zeitpunkten sich Menschen in einem Raum aufhalten. Beispielsweise werden in vielen gewerblich genutzen Gebäuden werden grundsätzlich alle Räume beheizt - unabhängig von tatsächlicher Anwesenheit von Personen. Eine Erkennung von menschlicher Anwesenheit ist bei der Optimierung also essenziell.

Da die direkte Messung von Anwesenheit über z.B. Infrarot- oder Kamerasensoren rechtlich problematisch ist, soll in dieser Arbeit untersucht werden, inwiefern Machine-Learning Algorithmen genutzt werden können, um eine genaue Erwartung über die Anwesenheit von Personen anhand von CO2-Werten in der Raumluft zu treffen.

Die Motivation der Optimierung der Gebäudeautomatisierung exisitert, da ein steigender CO2-Gehalt der Raumluft nachweislich mit einer Abnahme der menschlich kognitiven Leistung einhergeht. Mehrere Studien konnten belegen, dass sowohl sprachliche als auch

[1] [Umweltbundesamt, 2022]
[2] [Umweltbundesamt, 2020]

logisch- mathematische Fähigkeiten abnehmen, sobald der CO2-Gehalt der Raumluft bestimmte Werte überschreitet.
Um eine angemessene Datengrundlage zu schaffen, wurden in diversen Büro-Räumen der FH Aachen Temperatur-, Luftfeuchtigkeits-, Infrarot- und CO2-Sensoren angebracht, deren Messungen kontinuierlich auf einer Datenbank gespeichert wurden. Der Zeitraum der Messwerte begann Mitte 2021.
In allen Räumen sind täglich ein oder mehrere Personen im Rahmen eines ca. 8 stündigen Arbeitstages anwesend, weshalb die Temperatur-, Luftfeuchtigkeits- und CO2-Messwerte über einen Tag hinweg Schwankungen aufweisen. Diese sollen von einem Machine Learning Modell mit der direkten Präsenzmessung des Infrarotsensors gegenübergestellt werden, sodass das fertig Trainierte Modell anhand der Messwerte von Temperatur, Luftfeuchtigkeit und CO2 Aussagen über menschliche Präsenz treffen kann.
Die Frage, ob ein in einem Raum trainiertes Modell auch präzise Aussagen über die Anwesenheit in einem anderen Raum treffen kann, stellte eines besondere Bedeutung dar.

Es gab keine Einschränkungen hinsichtlich dessen, welche konkreten Machine-Learning Algorithmen benutzt werden sollen.
Als Programmiersprache für das Projekt wurde Python gewählt. Python ist wegen seiner umfassenden Machine-Learning Bibliotheken und einfachen Auswertungstechniken anhand von z.B. Graphen und Statistiken gut für diesen Anwendungsfall geeignet.

1.2. Vorgehensweise

Das Projekt beschäftigte sich im Schwerpunkt mit den folgenden Arbeitsschritten:

- Datenbeschaffung durch Datenbankzugriffe per SQL
- Analyse und Vorbereitung der Daten (Pre-Processing)
- Trainieren von Machine-Learning Modellen anhand der vorbereiteten Datensets
- Ergebnisauswertung durch Gegenüberstellung verschiedener Datensets und Models

Da es zwischen allen verfügbaren Datensets der einzlnen Räume und Machine-Learning-Models eine Vielzahl an Kombinationsmöglichkeiten gibt, war es ein Anspruch der Projektarbeit, ein möglichst übersichtliches, gut gekapseltes Python Programm zu erstellen, mit dem man einfach und schnell verschiedene Datensets verarbeiten und mit einer dem Forschungszweck angemessenen Anzahl von Machine-Learning Models auszuwerten.

Um einen Vergleich der Ergebnisse zu ermöglichen, sollen diese klar und verständlich dargestellt werden. Da nicht bei allen Algorithmen die gleichen Leistungsindikatoren genutzt werden, sollen hauptsächlich nur jene Indikatoren betrachtet werden, die bzgl. aller Algorithmen auch gleiche Bedeutung haben. Falls Model- spezifische Leistungsindikatoren als besonders Erkenntnisreich erachtet werden, wird dies in dieser Arbeit angemerkt.

2. Stand der Technik für Anwesenheitserkennung/-Prognose

Man kann verschiedene Techniken zur Anwesenheitserkennung und -Prognose in Innenräumen in aktive und *passive* Verfahren unterteilen.

Aktive Verfahren messen die menschliche Anwesenheit etwa mit Kamera- oder Infrarotsensoren, welche den Menschen direkt im Raum erkennen können. Ein Kamerasensor nimmt dabei kontinuierlich Bilder des Raumes auf, während ein Algorithmus im Hintergrund versucht ein 3D-Modell eines menschlichen Skeletts auf sich bewegende Teile des Bildes zu legen. Bewegen sich über mehrere Bilder hinweg die Bereiche auf und unmittelbar neben dem 3D-Modell, registriert das System eine Anwesenheit.

Infrarotsensoren dagegen funktionieren eher wie ein klassischer Bewegungsmelder, welcher Unterschiede in der Infrarotstrahlung eines Raumes erkennt. Durch die generierte Körperwärme des menschlichen Körpers, sorgt ein Mensch, der sich durch den Sensor bewegt, für einen Ausschlag des Sensors.

Passive Verfahren orientieren sich an den physikalischen Eigenschaften des Raumes. Anstatt die Präsenz des Menschen direkt zu messen, wird versucht, diese von Veränderungen der Eigenschaften der Raumluft oder elektromagnetischer Strahlung des Raumes zu erkennen.
Verändert sich beispielsweise durch menschliche Anwesenheit die Luftfeuchtigkeit in einem Raum in sehr geringem Maß, wird die Laufzeit einer Ultraschallwelle die durch diesen Raum geschickt wird, leicht verringert, da die Schallgeschwindigkeit in dichteren Medien zunimmt.
Die Systeme unterscheiden sich neben der Art der Messung auch deutlich in der Komplexität der Implementierung. Während Ultraschallsensoren wegen der verhätlnismäßig geringen Schallgeschwindigkeit keine besonders hohe Genauigkeit besitzen müssen, sind die an einen Mikrowellensensor gestellten Ansprüche wesentlich höher, da sich Mikrowellen mit $300 \cdot 10^6\, m/s$ gegenüber der Schallgeschwindigkeit mit lediglich etwa $330\, m/s$ ausbreiten. Mit Mikrowellen können Bewegungen durch Ausnutzung des Doppler-Effektes einer elektromagnetischen Welle erkannt werden. Bewegen sich Objekte oder Personen auf den Sensor zu, wird das Echo der Welle gestaucht, wodurch die Bewegung erkannt wird.
Hier verdeutlicht sich nochmals die Motivation dieser Arbeit, da Sensoren für Temperatur, Luftfeuchtigkeit, oder CO2 weder teuer sind, noch besondere Ansprüche an Positionierung im Raum oder Kallibrierung besitzen.

3. Untersuchte Verfahren

3.1. Machine Learning

Machine Learning ist ein Teilbereich der künstlichen Intelligenz und beschreibt das Entwickeln mathematischer Modelle zur statistischen Auswertung von Daten.
Dabei wird dem Modell anhand von Trainigsdaten beigebracht, Gesetzmäßigkeiten zu erkennen, um danach Erwartungen über bisher unbekannte Daten treffen zu können. Beispielsweise könnte ein solches Modell aus einem Datenset mit der aktuellen Jahreszeit, Uhrzeit und Position der Sonne am Himmel trainiert werden, sodass es auch schließlich in einem anderen Datenset aus Jahreszeit und Position der Sonne Rückschlüsse auf die Uhrzeit treffen kann.
Als Vorbild für diesen „Lernvorgang" dient das menschliche Gehirn, welches ebenfalls versucht zwischen bestimmten Input-Parametern wie z.B. der Form und Farbe eines Gegenstandes eine Beziehung herzustellen, um das beobachtete Objekt in Zukunft schneller kategorisieren zu können.
Da eine Vielzahl von effektiven Machine Learning Algorithmen existiert, ist es essenziell, sich mit den Stärken und Schwächen einzelner Herangehensweisen zu befassen.

In dieser Arbeit sollen zwei Unterkategorien des Machine Learning genauer betrachtet werden:

- *Supervised Learning*
- *Unsupervised Learning*

Supervised Learning bedeutet zwischen bestimmten Feldern eines Datensets eine Beziehung zu einem sog. Label herzustellen, welches als eine Art Ergebnis aus den Eingabewerten gesehen werden kann. Ein so traniertes Model kann dann neue, ihm vorher unbekannte Datensets, mit einem Label versehen - etwa wie in dem o.g. Beispiel wo Jahreszeit und Sonnenposition die Eingabewerte und die Uhrzeit das Label darstellen. Der Begriff „*supervised*" ergibt sich daraus, dass das Datenset, mit dem das Model traniert wird, diese Labels gegeben hat, sodass das Modell sich bei jedem Schritt des Lernvorgangs selbst korrigieren kann, falls eine Fehleinschätzung getroffen wurde. Bei einer sog. „*Klassifizierung*" sind diese Labels fest vorgegeben, während sie in der „*Regression*" kontinuierlicher Natur sind. Im Kontext dieser Arbeit wäre das Ergebnis einer Klassifizierung eine „1" für Anwesenheit und eine „0" für Abwesenheit, während das Ergebnis einer Regression eine Wahrscheinlichkeit auf Anwesenheit zwischen 0.0 und 1.0 darstellen würde.

Beim „*Unsupervised Learning*" versucht das Modell ohne Referenz zu einem bestimmten Label, Zusammenhänge zwischen bestimmten Feldern des Datensets herzustellen. Solche Modelle arbeiten vorrangig mit „*Clustering*" und „*Dimensionality Reduction*".
„*Clustering*"-Algorithmen versuchen ein Datenset in kleinere Teilmengen einzuteilen und so aus den Feldern des Datensets bestimmte Abhängigkeiten abzuleiten.

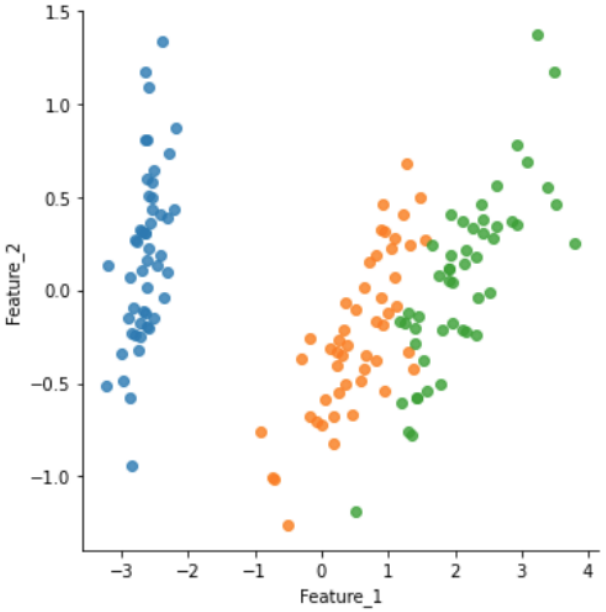

Abbildung 3.1.: Beispiel für Clustering

Bei der „*Dimensionality Reduction*" versucht der Algorithmus das Datenset in seiner Dimensionalität, also seiner Anzahl an Feldern, zu reduzieren. Es wird also die Frage gestellt, ob sich in einem bestehenden Datenset auch mit weniger Feldern Abhängigkeiten feststellen lassen. Dieser Schritt wird vorallem für Modelle benutzt, die sensibel gegenüber hoher Dimensionalitäten sind, sodass das Datenset vor dem Training in seiner Dimensionalität heruntergebrochen werden kann.
Im Rahmen des Projektes wurden hauptsächlich Klassifizierungs-Algorithmen genutzt, da ein Großteil der Datensets Labels zur Überprüfung hatte.
Um einen Vergleich herzustellen werden später trotzdem noch einzelne Ergebnisse von Clustering und Dimensionality Reduction betrachtet. Im Folgenden sollen die genutzten Modelle erklärt werden.

3.1.1. Random Forest Classifier

Random Forest Classifier RFC stellen eine Unterkategorien der „*Decision Trees*" dar. Decision Trees sind einfache Anordnungen von bestimmten Fragen, die über das Datenset gestellt werden, um eine Klassifikation zu erreichen.

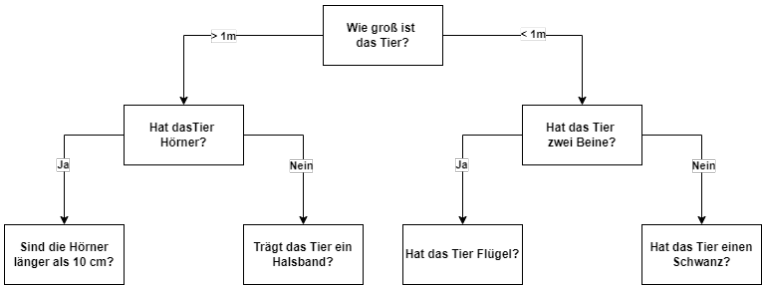

Abbildung 3.2.: Beispiel eines Decision Trees

Erstellt man ein „*Ensemble*" aus Decision Trees die Erwartungen über einen zufällig gewählten Teil des Datensets treffen können, entsteht ein Random Forest. Der Random Forest Classifier versucht, eine Menge einfacher Schätzfunktionen über einen komplexeren Sachverhalt „abstimmen" zu lassen. Während sich in einem einzelnen Entscheidungsbaum Fehleinschätzungen entwickeln können, sinkt die Chance auf eine solche Fehleinschätzung, je mehr unabhängige Entscheidungsbäume man befragt.

3.1.2. Gradient Boosting Classifier

Der *Gradient Boosting Classifier* GBC ist eine Abwandlung eines RFC und versucht, seine Erwartungen aufgrund von Abweichungen eines Labels vom Durchschnitt dieses Labels zu treffen.
Erweitert man das Datenset im o.g. Beispiel um das Alter einer Maus, wird ein GBC als Ausgangswert den Durchschnitt aller Label-Werte, also dem Gewicht, berechnen. Danach werden die Abweichungen aller Label-Werte zu diesem Durchschnitt gebildet. Diese Abweichungen werden nun in Beziehung zu den anderen Spalten des Datensets gesetzt.

Beispielsweise könnte man so davon ausgehen, dass ausgewachsene Mäuse von einem bestimmten Alter über dem Durchschnittsgewicht liegen. Genauso liegen besonders junge Mäuse wahrscheinlich immer einen ähnlichen Wert unter dem Durchschnittsgewicht. So wurde zwischen dem Label *Gewicht* und der Spalte *Alter* eine Beziehung hergestellt. In weiteren Iterationen orientiert sich der GBC immer an der Abweichung zum Durchschnittswert des vorherigen Baumes. So werden die getroffenen Erwartungen über mehrere Iterationen immer präziser.

3.1.3. Support Vector Classifier

Der *Support Vector Classifier* SVC versucht in einem Datenset anhand von bestimmten Cut-Off-Values klare Grenzen zwischen Werten zu finden, sodass man alle Messwerte ober- und unterhalb der Grenze eindeutig Klassifizieren kann.

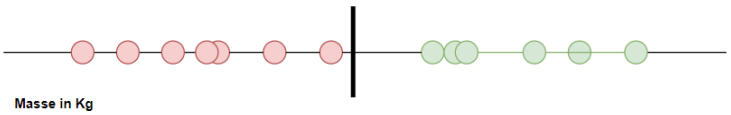

Abbildung 3.3.: Beispiel eines Support Vector Classifiers

In Abb. 3.3 ist der SVC ein Punkt auf einer eindimensionalen Linie, auf der das Gewicht in Kg von z.B. Mäusen in „Unter-" und „Übergewichtig" unterteilt wird. Dieser Punkt ist Ergebnis aller Verhältnisse der einzelnen Datenpunkte zueinander. Durch sog. „*Kernel Funktionen*" versucht der Algorithmus nun Beziehungen in höheren Dimensionen zu finden, wie z.B. $Masse^2$, $Masse^3$ usw. . Der SVC stellt dann in diesen Dimensionen eine Linie in einem zwei-dimensionalen oder eine Ebene in einem drei-dimensionalen Koordinatensystem dar.

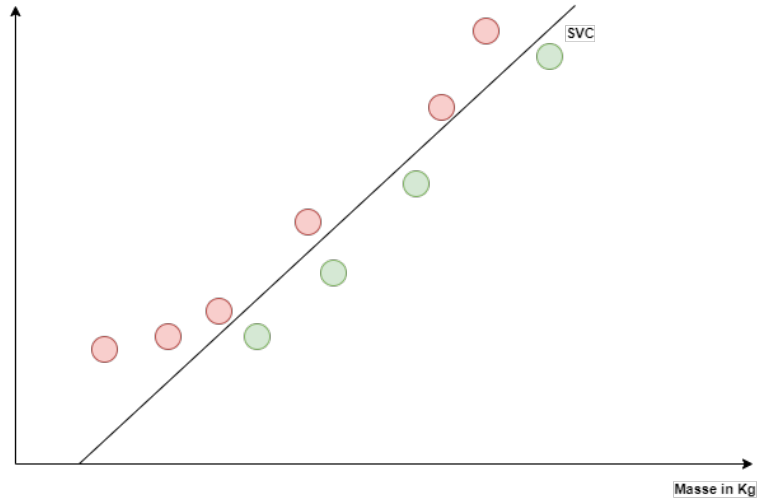

Abbildung 3.4.: Beispiel eines Support Vector Classifiers in der zweiten Dimension

Da der SVC die Verhältnisse aller Datenpunkte zueinander betrachtet, ist er sehr anfällig für Ausreißer in den Daten, was bei der Datenvorbereitung und der Auswertung beachtet werden muss.

3.1.4. Logistische Regression

Die logistische Regression ordnet Daten einem bestimmten Label anhand einer Wahrscheinlichkeit zu. Trotz dem Teilnamen „Regression" handelt es sich um einen Klassifizierungsalgorithmus. Der Name kommt daher, dass die berechnete Wahrscheinlichkeit zu einem kontinuierlichen Label zwischen 0 und 1 führt, was durch die Form einer Sigmoid-Funktion ausgedrückt wird. Der Algorithmus eignet sich wegen dieser Eigenschaft besonders für das Klassifizierungsproblem der Projektarbeit, bei der ein Wahrheitswert, wie „Anwesenheit" oder „Abwesenheit" untersucht werden soll.
Nutzt man logistische Regression beispielsweise zur Abbildung einer Beziehung zwischen Gewicht von Mäusen in Gramm zu einer Wahrscheinlichkeit für Übergewicht, könnte das entstehende Modell wie folgt aussehen.

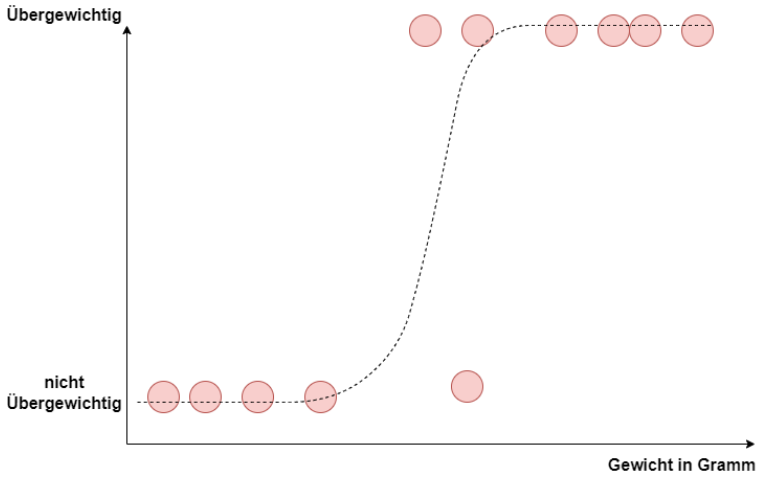

Abbildung 3.5.: Beispiel logistischer Regression

Die Sigmoid-Funktion wird grundsätzlich durch

$$p = \frac{1}{1 + e^{-(x-\mu)/s}} \tag{3.1}$$

beschrieben, wobei x der Eingabewert in Gramm und μ der Mittelpunkt der Kurve beschreibt, der sich aus dem Verhältnis von Eintrittswahrscheinlichkeit p_1 und Gegenwahrscheinlichkeit p_0 ergibt.

$$p(\mu) = 0.5 = \frac{p_1}{p_0} \tag{3.2}$$

Zusätzlich beschreibt s einen Skalierungsparameter, mit dem die Form der Kurve flacher oder steiler werden kann, was angibt, wie eindeutig die hereingegebenen Daten zugeordnet werden können. Wären im o.g. Beispiel also alle roten Punkte jeweils links und rechts von der Mitte einsortiert, wäre die Sigmoid-Funktion in der Mitte sehr steil, da alle Daten anhand des Mittelpunktes klar getrennt werden könnten.

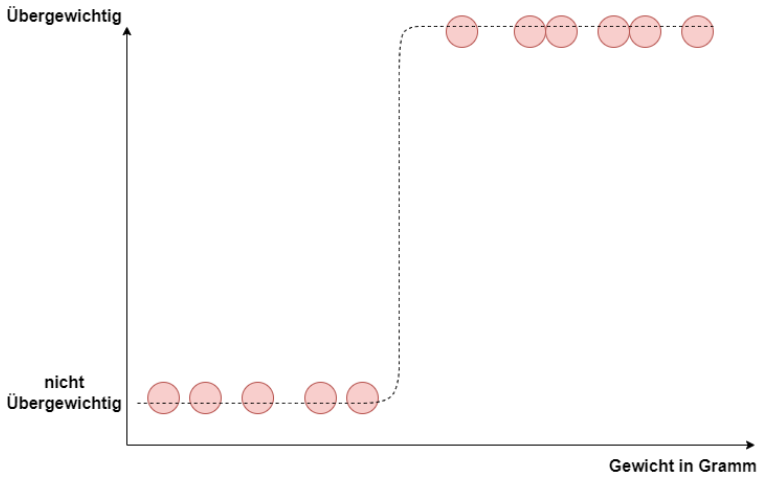

Abbildung 3.6.: Logistische Regression mit deutlicher Skalierung

Anhand der optimierten Sigmoid-Funktion können nun neue Daten klassifiziert werden, wobei die Wahrscheinlichkeit bei $p < 0.5$ auf 0 und $p >= 0.5$ auf 1 gerundet wird.

3.1.5. K-Nearest-Neighbours

Der *K-Nearest-Neighbours*-Algorithmus KNN betrachtet den Abstand von einem gegebenen Punkt p zu einer Menge an Punken in einem Datenset und ordnet ihn gemäß dieser Abstände einer bestimmten Kategorie zu. Die Zuordnung geschiet anhand der Kategorie der größten Menge an nähesten Nachbarn.

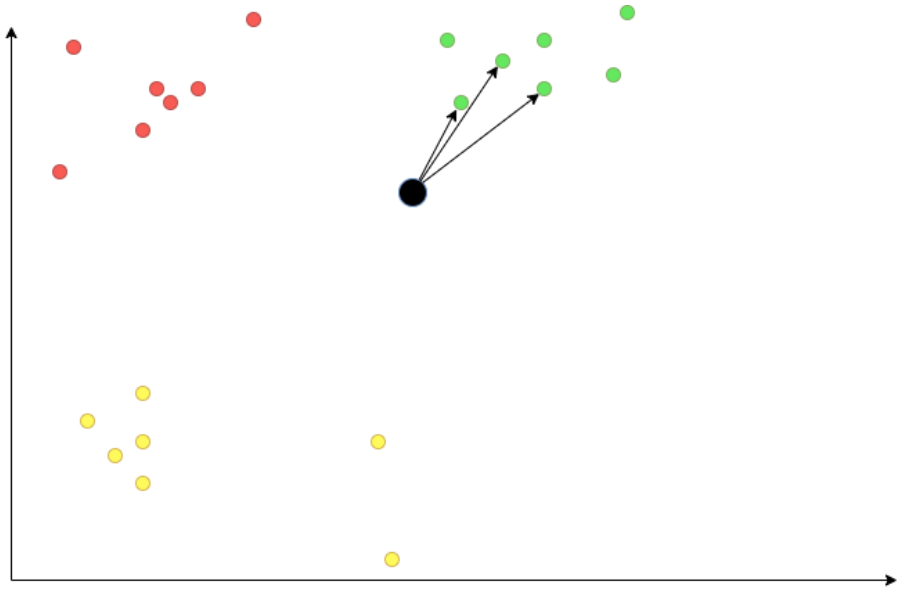

Abbildung 3.7.: Beispiel eines KNN

Im oben gezeigten Beispiel würde ein Punkt anhand seiner drei nächsten Nachbarn als ”grün”kategorisiert. Die Anzahl der gewünschten Nachbarn die zu untersuchen sind, ist frei wählbar. Da die Abstandsberechnung über den *Euklidischen Abstand* berechnet wird, funktioniert dieser Algorithmus auch in höheren Dimensionen, da

$$d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2 + ... + (n2 - n1)^2} \tag{3.3}$$

sich in beliebig viele Dimensionen fortsetzen lässt.

3.1.6. Neuronale Netzwerke

Die Funktionsweise eines Neuronalen Netzwerks ist direkt angelehnt an die Funktionsweise des menschlichen Gehirns. Einzelne Knotenpunkten(Neuronen) werden mithilfe von Gewichteten Verbindungen verknüpft, sodass das Netzwerk versucht Eingabewerte bestimmten Ausgabewerten zuzuordnen. Diese Zuordnung der Ein- und Ausgabewerte im Input- und Output-Layer geschiet nicht direkt, sondern durch ein oder mehrere *Hidden Layer*, dessen Neuronenzahl üblicherweise über der Anzahl Neuronen im Input Layer liegt. Die Anzahl der Neuronen im Output-Layer entspricht der Anzahl an Ergebnissen, die sich aus dem Input Ergeben können. Im Beispiel der Anwesenheitsanalyse entspräche das hier also zwei Neuronen für An- und Abwesenheit.

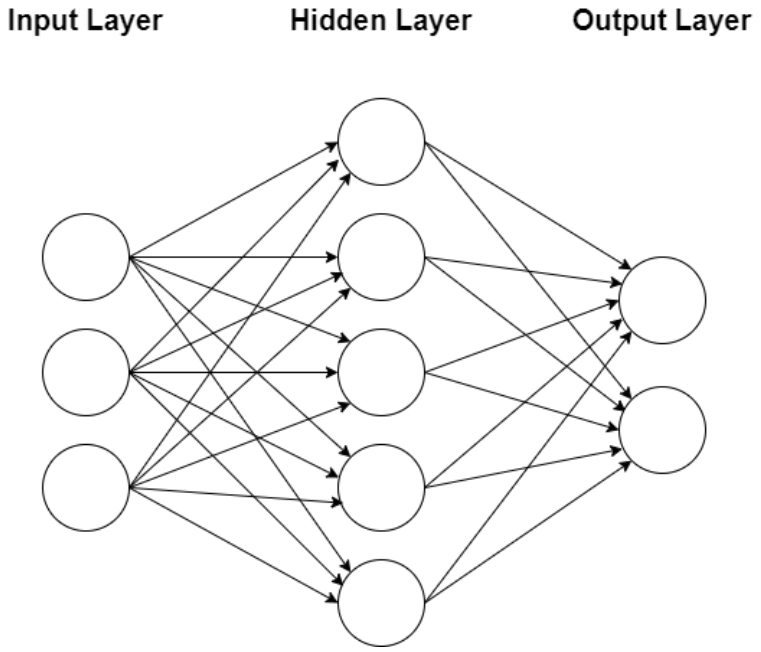

Abbildung 3.8.: Beispiel eines neuronalen Netzwerks

Liegt an einem Neuron eine Information an, wird diese als Eingabe einer Aktivierungsfunktion φ genutzt, die mithilfe eines bestimmten Schwellwertes bestimmt, ob dieses Neuron aufgrund der Eingabe aktiviert wird. Über eine bestimmte Anzahl von Iterationen werden die Gewichtungen zwischen den einzelnen Neuronen stärker oder schwächer.

3.1.7. Long Short Term Memory

Das *Long Short Term Memory* LSTM ist eine Abwandlung herkömmlicher Neuronalen Netzwerke. Es handelt sich um ein *rekurrentes* Neurales Netzwerk, was bedeutet, dass jedes Neuron seine Ausgabewerte auch wieder als Eingabewerte nutzt. Der Begriff *Memory* rührt daher, dass durch diese Rückkopplung eine Art Gedächtnis entsteht, durch die das Netzwerk bessere Rückschlüsse auf die Einordnung des aktuellen Input-Wertes ziehen kann.

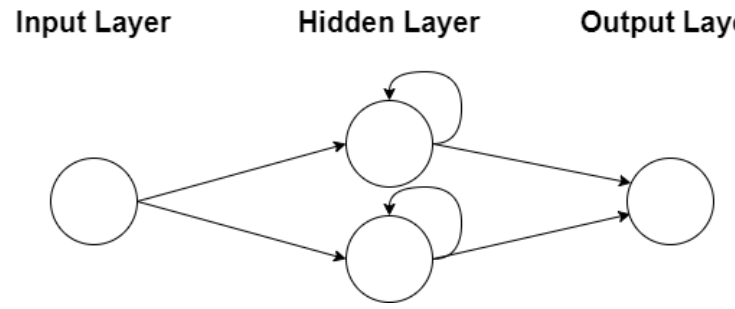

Abbildung 3.9.: Beispiel eines rekurrenten neuronalen Netzwerks

Bei einem LSTM verfügt jedes Neuron zusätzlich über einen Zell-Status C_t, welcher durch einen Input x_t verändert werden kann. Auf den Zell-Status nehmen *Input*-, *Forget*- und *Output*-Gates Einfluss. Diese drei Gates besitzen jeweils ein eigenes neronales Netzwerk σ.

Das Input-Gate bestimmt, ob der Zell-Status anhand des anliegenden Inputs verändert werden darf. Das Forget-Gate gibt an, ob der Status der Zelle zurückgesetzt und somit das „Gedächtnis" der Zelle gelöscht wird. Es stellt die zentrale Komponente des LSTMs dar. Das Output-Gate regelt, ähnlich wie Aktivierungsfunktion normaler NNs, ob der anliegende Input zu einem Output der Zelle führt.

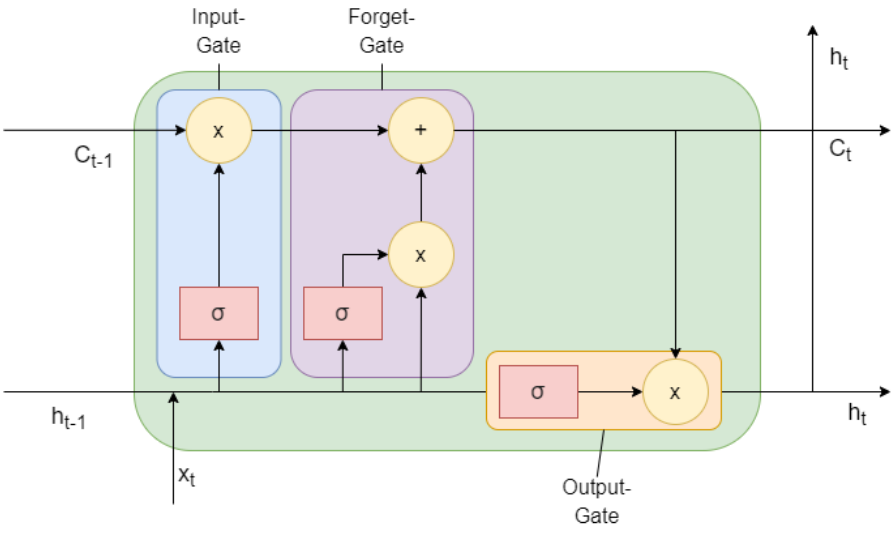

Abbildung 3.10.: Vereinfachter Aufbau einer LSTM-Zelle

Ein LSTM kann auf diese Weise eine Vielzahl von Zeitschritten zurückblicken und verlässt sich so nicht direkt auf einen gegebenen Input, sondern auf einen langen Verlauf von bereits verarbeiteten Input-Werten. LSTMs sind deshalb besonders interessant für Probleme bei denen Beziehungen zwischen kontinuierlichen Datenwerten gebildet werden müssen.

Geht man von einem LSTM aus, dass über eine Liste an Namen verfügt und anhand einem Datensatz gelernt hat, zwei Namen mit der Beziehung „sah" zu z.B. „Jonas sah Jakob." zu verknüpfen.
Das LSTM hat nun intern einen Status, der angibt, dass auf einen Namen mit hoher Wahrscheinlichkeit das Wort „sah" und mit niedriger Wahrscheinlichkeit ein zweiter Name folgt. Beginnt man nun einen neuen Satz mit „Jakob", werden als mögliche nächste Worte „Jonas", „Jakob" und „sah" ausgewählt. Bisher ähnelt dieser Verlauf einem normalen Neuronalen Netzwerks. Beim LSTM steht aber nun wegen dem vorherigen Durchlauf der Name „Jakob" im Forget-Gate, sodass „Jakob" aus der verfügbaren Wortauswahl für das nächste Wort gelöscht wird.

3.2. CO2 als Anwesenheitsindikator

Der Anteil von CO2 in frischer Atemluft beträgt zwischen 350 und 450 ppm. Es gibt in Deutschland und auch Europa keine grundsätzlich festgelegten Grenzwerte für akzeptable Raumluft, vielmehr raten Gesundheitsämter verschiedener Länder Grenzwerte zwischen 1200 und 1500 ppm einzuhalten. Bei der Obergrenze von 1500 ppm entstehen beim Menschen erste Müdigkeitserscheinungen, weshalb dieser Wert in der Literatur als maximaler Richtwert für Innenräume gilt.

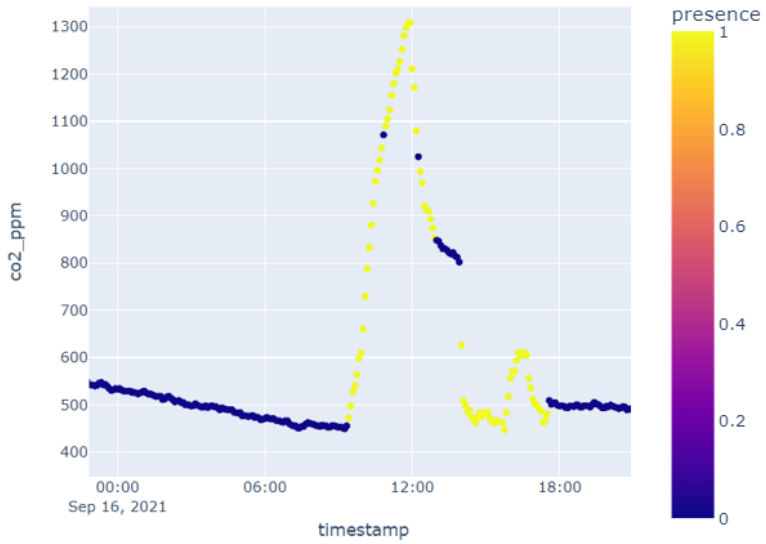

Abbildung 3.11.: CO2 Gehalt der Raumluft über einen Tag

Es ist zu erkennen, dass die CO2-Werte in einem normalen Büroraum innerhalb dieser empfohlenen Grenzen schwanken. Mit einigen kleinen Pausen sind fast durchgehend Personen anwesend, die täglich etwa um 12:00 den Raum lüften. Es ist auch klar zu erkennen, dass sich der CO2-Gehalt während Abwesenheit durch die passive Lüftung des Raumes (Tür-/Fensterspalten) langsam wieder gegen den Grundwert bewegt.

Tabelle 3.1.: Sensordaten

Sensordaten		
Name	Format	Beschreibung
timestamp	timestamp	Zeitpunkt der Messung
co2_ppm	integer	CO2-Wert
temperature_celsius	float	Temperatur in Grad Celsius
relative_humidity_percent	float	Luftfeuchtigkeit
presence	boolean	Aktivität Bewegungssensor

3.3. Luftfeuchtigkeit und Temperatur

Auch wenn Luftfeuchtigkeit und Temperatur Indikatoren für menschliche Präsenz in einem Raum sein können, unterliegen sie in ihrer Nützlichkeit dem CO2-Gehalt der Raumluft in einem wichtigen Faktor. Sie sich beide stark beeinlusst, von äußeren Umweltfaktoren wie dem aktuellen Wetter oder der Jahreszeit. Beide Werte wurden zunächst in das Training aller Modelle miteinbezogen, es wird allerdings im Folgenden Kapitel gezeigt, dass diese beiden Werte zusammen mit dem CO2-Werte nicht von besonderem Nutzen sind und deshalb später nicht weiter genutzt werden.

3.4. Sensordaten

Wie bereits beschrieben, wurden die Sensordaten in mehreren Räumen der FH Aachen kontinuierlich gesammelt. Durch das Filtern nach der Beziehung des Raumes ergab sich folgende Datenstruktur als Ausgangslage:

4. Technische Umsetzung

4.1. Datenbeschaffung und -Vorbereitung

Die Daten wurden lokal auf einem der FH-Server in Form eines *Hadoop Distributed File System* (HDFS) gespeichert. Mithilfe von Apache Drill konnten die Daten jederzeit mit einfachen SQL-Abfragen beschafft werden. Die Daten wurden so durch die gesamte Dauer des Projektes immer aktuell gehalten, damit alle Erkenntnisse immer auf der aktuellsten Datenlage basieren.

Die Datenvorbereitung oder Pre-Processing ist eine der wichtigsten Schritte bei der Anwendung von Machine Learning. Durch sie kann man beim Training des Models durch Bearbeitung bestehender Spalten oder Hinzufügen von zusätzlichen Spalten im Datenset Schwerpunkte setzen, die es den Algorithmen beim Training zum einen erleichtern, ihre Erwartungen zu präzisieren, zum anderen aber auch die Leitung beim Verarbeiten bestimmter Spalten zu steigern.

4.1.1. Gruppierung

Die Sensordaten wurden alle sechs Sekunden erfasst. Da sich weder CO2-Gehalt noch Feuchtigkeits- oder Temperaturwerte der Raumluft so schnell nicht verändert, wurden die Daten direkt beim Drill per SQL zu zwei-Minuten-Intervallen zusammengefasst. Dabei werden über alle Spalten hinweg Durchschnittswerte gebildet, die dann nachher zu einem Datensatz zusammengefasst werden. Dies steigert die Leistung aller Algorithmen erheblich, da sich die Zahl der Datensätze sich stark verringert. Da sich, wie oben erwähnt, der CO2-Gehalt der Raumluft in einem Intervall von zwei Minuten kaum merklich verändert, verringert sich die Genauigkeit des gesamten Datensetz dadurch nicht maßgeblich.

4.1.2. Zyklische Codierung

Zyklische Codierung wird immer dort verwendet, wo Daten sich in wiederholenden Schemata bewegen. Diese Schemata, wie z.B. die Zahlenumbrüche bei einer Uhrzeit, sind für Algorithmen nicht direkt ersichtlich und sind zudem für Computer nicht leicht zu verarbeiten. Durch eine Encodierung in Sinus- und Cosinus-Werte können diese Zusammenhänge vereinfacht werden.
Hierzu wurde der Timestamp zuerst in Sekunden übersetzt, sodass sich ein bestimmter Zeitpunkt eines Tages immer zwischen 0 und 86400 Sekunden bewegt. Aus diesem Wert wurden dann zwei neue Datenspalten „*hour_sin*" und „*hour_cos*" in das Datenset eingefügt welche sich durch

$$hour\_sin = sin(2 * \pi * x/x_{max}) \tag{4.1}$$

$$hour\_cos = cos(2 * \pi * x/x_{max}) \tag{4.2}$$

ergeben. So kann jede Tageszeit einer eindeutigen Kombination aus Sinus- und Cosinus-Werten zwischen 0 und 1 zugeordnet werden.

4.1.3. Deltas und Shift-Werte

Desweiteren wurden von den Spalten „*co2_ppm*", „*temperature_celsius*" und „*relative_humidity_percent*", die tatsächlich Rückschlüsse auf die Präsenz zulassen, zusätzliche Delta- und Shift-Spalten angelegt.

Ein Shift-Wert bedeutet lediglich, dass in einer Zeile x_n des Datensets zusätzlich, neben den aktuellen Werten, auch Werte von k Zeilen zuvor, also x_{n-k} stehen. So haben alle Algorithmen direkten Zugriff auf Vergangenheitswerte der ausgewählten Spalten.

Delta-Spalten stellen, dem Namen nach, Deltas zu vorherigen Werten dar:

$$\Delta x_k = x_n - x_{n-k} \tag{4.3}$$

Die Erwartung ist hier, dass die Änderung der CO2-, Temperatur- und Luftfeuchtigkeitswerte ein wichtigerer Indikator sein könnte, als die tatsächlichen Werte. In einem schlecht klimatisierten Raum könnten Grundwerte von z.B. CO2 höher sein, als in anderen Räumen. Durch die hinzufügten Deltas werden diese Grundwerte ignoriert und Rückschlüsse auf die aktuelle Präsenz sind besser möglich.
Im Zuge der Projektarbeit wurden verschiedene Kombinationsmöglichkeiten von Delta- und Shiftwerten mit Zeitschritten zwischen zwei Minuten und einer Stunde mit Hinblick auf Verbesserungen der Model-Genauigkeiten getestet.

4.1.4. Outlier Detection

Wie bereits erwähnt, spielen Datenausreißer für die Ergebnisse mancher Algorithmen eine große Rolle. Überall wo z.B. aus einer Reihe von Datenwerten Durchschnittswerte berechnet werden, würden Ausreißer in den Daten das Ergebnis verfälschen und die Leistung des Algorithmus deutlich senken. Um diese Ausreißer vor dem Training der Models zu beseitigen wurde das Verfahren des *Interquartilabstands* (IQR nach der englischen Bezeichnung *Interquartile Range*) gewählt.
Der IQR gibt die Intervallgröße an, die ein Wert vom Median einer Datenreihe abweichen darf. Bei einer der Größe nach sortierten Datenreihe $x = (x_0, x_1, ..., x_n)$ bestimmt man die Mediane der unteren und oberen Hälfe des Datensets Q_1 und Q_2. Der IQR ergibt sich nun aus

$$IQR = Q_2 - Q_1 \tag{4.4}$$

Mit diesm Wert kann man nun die erlaubten Ober- und Untergrenzen des Datensets mit

$$Limit_{upper} = Q_2 + 1.5 * IQR \tag{4.5}$$
$$Limit_{lower} = Q_1 - 1.5 * IQR \tag{4.6}$$

bestimmen. Alle Werte die außerhalb dieser Grenzen liegen, können als Ausreißer betrachtet werden.

Ausreißer zu entfernen, ist hier wichtig, da die Sensoren Messfehler erzeugen können, oder gelegentlich zur Demonstration von starken Veränderungen in der direkten Nähe des Sensors geatmet wurde und es so in vielen Datensets kurzfristige CO2-Werte gibt, die natürlich in geschlossenen Räumen nicht vorkommen.

4.2. Validierung

Über das Projekt hinweg wurden verschiedene Validierungsmethoden verwendet. Grundsätzlich unterteilt man das Datenset in ein Trainings- und Testset mit einem Verhältnis von etwa 80-20. Das bedeutet, dass das Model mit 80 prozent der Daten trainiert wird, wobei die anderen 20 prozent zurückgehalten werden, um daraufhin das fertig trainierte Model daran zu testen. Aus dem Ergebnis dieses Tests ergeben sich verschiedene Werte, die die allgemeine Genauigkeit des Models repräsentieren, d.h. wie verlässlich das Model die An- und Abwesenheit des Testsets selbst berechnen kann, wenn man ihm den tatsächlichen Wert vorenthält.
Allgemeine Basis für die errechneten Genauigkeiten bildet die sog. *Confusion Matrix*.

Confusion Matrix		Predicted	
		Negative	Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

Abbildung 4.1.: Beispiel einer Confusion Matrix

Diese zeigt, inwiefern sich die Genauigkeit des Modells neben tatsächlich richtig erkannten Werten zusätzliche aus false negatives und false positives zusammensetzt.

Die *Accuracy Score* ist eine einfache Methode zur Bestimmung der Modellqualität und berechnet sich aus der Summe der richtigen Ergebnisse geteilt durch die Summe aller Ergebnisse.

$$AccuracyScore = \frac{TP + TN}{TP + TN + FP + FN}$$

Die *Recall Score* beschreibt die Genauigkeit bezogen auf alle erkannten positiven Ergebniswerte.

$$RecallScore = \frac{TP}{TP + FN}$$

Die *Precision Score* gibt die allgemeine Menge, an positiven Werten, die hätten erkannt werden müssen, an.

$$PrecisionScore = \frac{TP}{TP + FP}$$

Die *F1-Score* ist der Durchschnittswert aus Recall- und Precision-Score und gibt im Allgemeinen eine gute Auskunft über die Qualität eines Modells.

$$F1\_Score = \frac{2}{\frac{1}{PrecisionScore} + \frac{1}{RecallScore}} = \frac{2 \cdot (PrecisionScore \cdot RecallScore)}{PrecisionScore + RecallScore}$$

In diesem Anwendungsfall ist es durchaus wichtig, Anwesenheitswerte genauer zu betrachten als Abwesenheitswerte, da ein Arbeitstag nur etwa ein Drittel eines tatsächlichen Tages darstellt. Das Datenset ist also mit einem Verhältnis von etwa 8/16 in Richtung der Abwesenheit unausgeglichen. Dies lässt die Erwartung zu, dass das trainierte Model wesentlich besser darin sein wird, Abwesenheit zu erkennen, als Anwesenheit.
Hierzu wurde bei der Auswertung der Ergebnisse immer auch der *Classification Report* hinzugezogen, aus dem ersichtlich ist, wie genau das Model beide möglichen Labelwerte berrechnen konnte.

Desweiteren war es entscheidend zu erkennen, dass das Datenset über mehrere Monate hinweg nicht perfekt uniform ist, da es in bestimmten Monaten zu verhätlnismäßig vielen Urlaubstagen (z.B. Weihnachten/Silvester) und damit einer Häufung an Abwesenheits-Werten kommt. Wenn die Daten nun im o.g. Verhältnis aufgeteilt werden und dann viele Daten des Testsets in solchen Monaten liegen, könnte es beim Ergebnis zu Verzerrungen kommen.
Hierfür wurde eine *Kreuzvalidierung* (engl. Cross Validation CV) implementiert. Bei einer CV wird das Datenset immernoch im gleichen Verhältnis aufgeteilt, allerdings mehrmals, sodass die Trainings- und Testsets jedes mal aus jeweils anderen Daten bestehen.

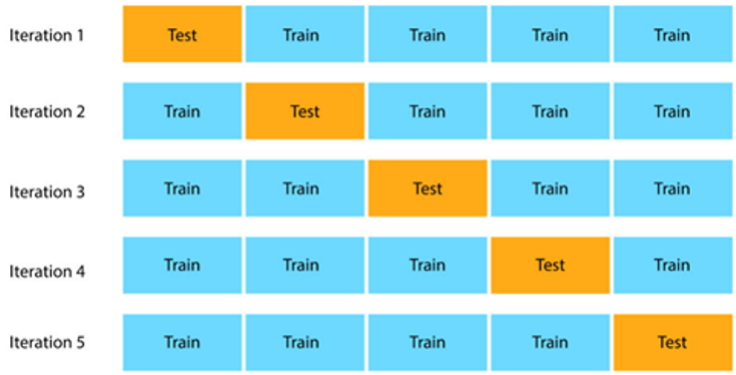

Abbildung 4.2.: Beispiel einer Kreuzvalidierung

Errechnet man nun die Durchschnittsgenauigkeit aller Iterationen der CV, ergibt sich eine Gesamtgenauigkeit, die das Model besser repräsentiert, weil es an einer größeren Anzahl an verschiedenen Daten trainiert und getestet wurde.

Da es für manche Räume keine Datensätze mit Labelwerten gab, musste die Validierung dieser Datensätze augenscheinlich erfolgen. Hierfür wurde in ein Datenset eine Labelspalte eingefügt, die dann vom trainierten Model selbst gefüllt werden sollte. Das Ergebnis musste dann in einem Graphen gezeichnet und per Hand validiert werden. Da sich das Ergebnis der Berechnung in diesem Anwendungsfall, wie oben gezeigt, sehr übersichtlich als Graph darstellen lässt, war diese Methode der Validierung zwar nicht perfekt, lieferte aber trotzdem einen ausreichenden Eindruck über die Qualität des trainierten Models.

4.2.1. Over- und Underfitting

Over- und Underfitting sind Probleme, die bei allen Machine Learning Algorithmen auftreten können und bezeichnen im Allgemeinen einen falschen Umgang mit dem Datenset. *Overfitting* bedeutet, dass das Modell mit zu vielen, sich zu stark ähnelnden, Daten trainiert wurde, wodurch es nur in der Lage ist, neue Daten richtig zu kategorisieren, die dem Trainingsset besonders ähnlich sind. Overfitting kann während einer Kreuzvalidierung oder Validierung anhand eines anderen Datensets, als dem Trainingsset, erkannt werden.

Um Overfitting zu verhindern, kann das Datenset zu vereinfacht werden, um so die Beziehungen zwischen den für die Erwartungsberechnung relevanten Spalten zu stärken.

Underfitting bedeutet dagegen, dass es das Modell nicht geschafft hat, zwischen den gegebenen Daten Beziehungen herzustellen, die das Modell unbekannte Daten richtig kategorisieren lassen. Hier reicht es normalerweise aus, dem Modell mehr Daten zur Verfügung zu stellen und in der Datenvorbereitung Felder einzufügen, die dem Modell bestimmte Beziehungen zwischen Datenfeldern hervorheben sollen.

4.2.2. Parameter Tuning

Parameter Tuning beschreibt einen Schritt der Modell-Optimierung, der normalerweise stattfindet, nachdem ein funktionierendes Modell, das mit bereits verarbeiteten Daten gute Ergebnisse liefert, erstellt wurde. Jedes Modell besitzt Parameter, die bei der Erstellung festgesetzt werden. Diese Parameter haben großen Einfluss auf den Trainingsprozess, weshalb es sich anbietet, die Genauigkeiten mehrerer Modelle mit verschiedenen Paramter-Kombinationen zu testen.
Das im Scikit-Learn enthaltene *GridSearchCV*-Modul bietet die Möglichkeit einem Modell eine Vielzahl an verschiedenen Parameter-Optionen zu übergeben. Mit diesen werden kann automatisch Modelle trainiert und per Kreuzvalidierung verglichen. Am Ende liefert das Modul die Parameter-Kombination, mit der das beste Ergebnis erzielt wurde.

```
param_test = {
    'n_estimators':[25, 50, 100, 250],
    'max_depth': [2, 5, 10, 15],
    'min_samples_split':[25, 150, 500],
    'min_samples_leaf':[50, 75, 125],
    'max_features':[5, 6, 7, 8, 9],
    'subsample':[0.75, 0.9, 0.95]
     }
```

Abbildung 4.3.: Beispiel einer Sammlung von Parameter-Optionen

Im obenstehenden Beispiel kann man erkennen, dass für bestimmte Parameter (rot) eine Sammlung an erlaubten Werten (grün) übergeben werden. Alle Kombinationen aus diesen Parameter-Optionen werden dann vom GridSearchCV-Modul ausgewertet.
Die Verbesserungen gegenüber Standard-Parametern bewegen sich normalerweise im niedrigen einstelligen Prozent-Bereich und kann auch zu einer leichten Verschlechterung führen, wenn die Standard-Parameter der einzelnen Modelle bereits gut gewählt waren.

5. Anwendung und Ergebnisse

5.1. Feature Vektor

Der Feature Vektor FV beschreibt das Ergebnis des Datensets nach der Vorverarbeitung. Durch die o.g. Schritte ergibt sich eine Vielzahl an möglichen zusätzlichen Features, mit denen man das Datenset versehen könnte. Es sollte zunächst ermittelt werden, welche diese Features wirklich aussagekräftig sind, um den FV nicht unnötig zu überladen. Einige Algorithmen geben nach dem Training Auskunft darüber, in welchem Maß ein Feature in die Berechnung einfließt. Zur Wahl eines FVs der nach dem Training mit einem Modell eine möglichst hohe Genauigkeit liefert, wurden verschiedene Kombinationen durch Verknüpfung von Temperatur, Luftfeuchtigkeit, CO2 mit jeweiligen Delta- und Shiftwerten vorgenommen. Ziel war es, mit einem möglichst kleinen Vektor eine möglichst hohe Genauigkeit zu erreichen, da die meisten Algorithmen eine direkte Abhängigkeit zwischen Dauer der Berechnung und Dimensionalität des FV haben.

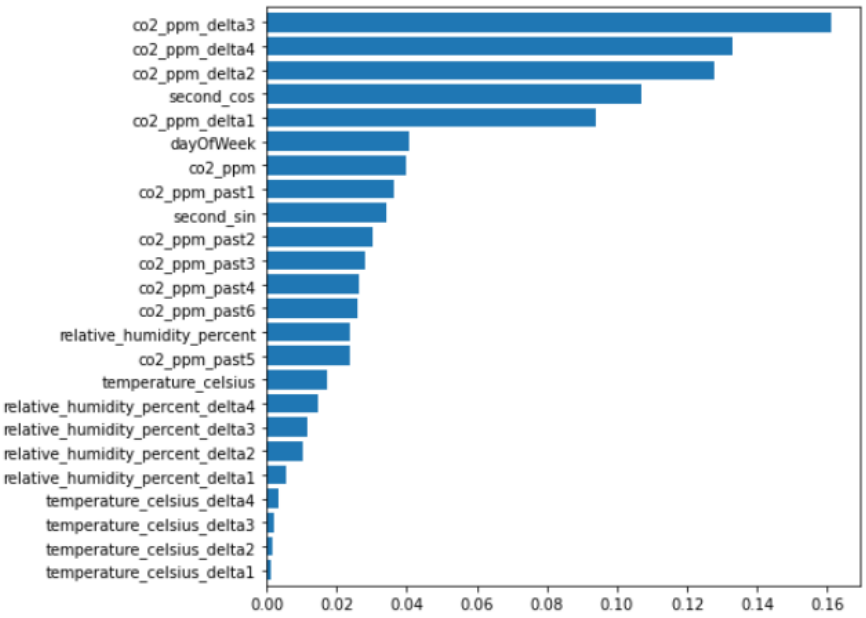

Abbildung 5.1.: Feature Importances eines Random Forest

In der o.g. Grafik sind neben den ursprünglichen Datenwerten nun auch Delta- und Shiftwerte zu sehen. Wie bereits im vorherigen Kapitel angedeutet, sieht man hier klar, dass die Temperatur- und Luftfeuchtigkeitswerte für die Errechnung der Labels kaum zu Rate gezogen werden. Weder die Grund- noch Deltawerte weisen eine hohe Wichtigkeit für das Model auf.

Nach Exkludierung von Temperatur und Luftfeuchtigkeit wurden neue Deltas und Shifts für die CO2-Werte eingefügt und die Feature Importances erneut betrachtet. Es ist zu erkennen, dass auch die CO2-Werte von n Minuten zuvor ebenfalls nicht maßgeblich in die Berechnung einfließen.

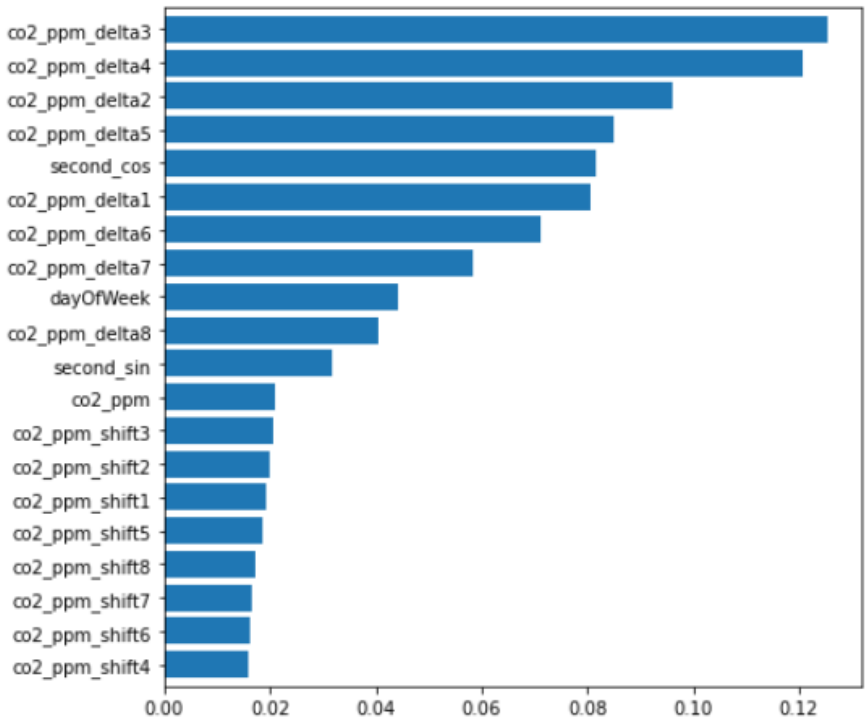

Abbildung 5.2.: Feature Importances eines Random Forest

Der Graph bestätigt die Annahme, dass beim CO2 die Deltawerte deutlich ausschlaggebender sind, als die tatsächliche Messung zu einem bestimmten Moment.

Wenn diese Werte nur in solch geringem Maß zur Berechnung beitragen, muss ebenfalls die Schlussfolgerung daraus, dass die Genauigkeit auch ohne diese Werte innerhalb einer gewissen Toleranz konstant bleibt, überprüft werden.

Tabelle 5.1.: Vergleich Feature Vektoren

Feature Vektor	Beschreibung
FV1	CO2, Temperatur und Luftfeuchtigkeit mit Shift-Werten
FV2	CO2, Temperatur und Luftfeuchtigkeit ohne Shift-Werte
FV3	Nur CO2 mit Shift-Werten
FV4	Nur CO2 ohne Shift-Werte

Im Folgenden werden beispielhaft vier verschiedene Feature Vektoren beschrieben, aus denen der Vektor, der im weiteren Verlauf des Projektes benutzt werden sollte, abgeleitet wurde.

Mit diesen vier Vektoren wurden nun verschiedene Modelle trainiert und deren Genauigkeiten gegenübergestellt.

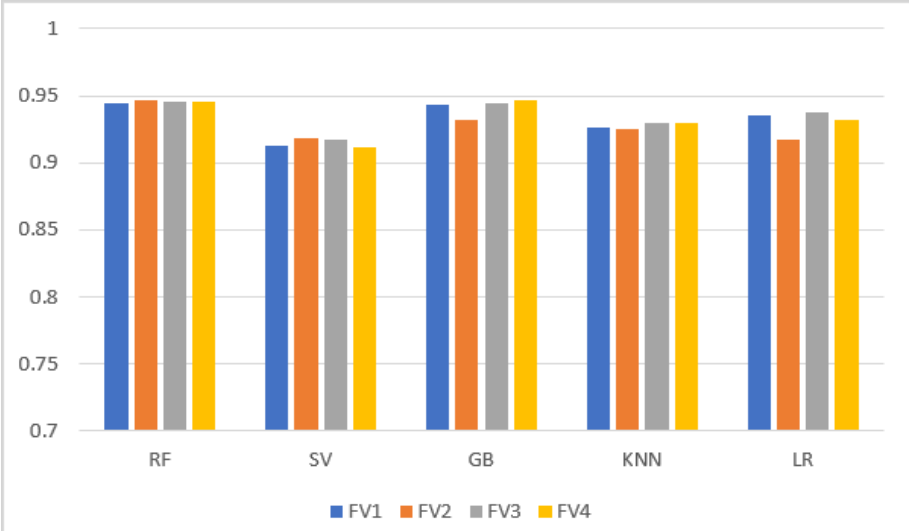

Abbildung 5.3.: Vergleich der Genauigkeiten

Wie deutlich sichtbar ist, sind die Genauigkeiten fast identisch. Diese Gegenüberstellung wurde über das Projekt hinweg mehrfach wiederholt und ergab immer ähnliche Unterschiede.
Dies war Anlass sowohl die Temperatur- und Luftfeuchtigkeitswerte, als auch die Shift-Werte der CO2-Messungen nicht weiter zu betrachten, um den FV für Modelle mit hoher Empflindlichkeit gegenüber Dimensionalität zu optimieren.

Tabelle 5.2.: Feature Vektor

Feld	Beschreibung
second_sin	timestamp-Sinusanteil
second_cos	timestamp-Cosinusanteil
day_of_week	Wochentag der Messung
co2_ppm	CO2-Wert
co2_ppm_delta1	Delta zum CO2-Wert vor 2 Minuten
co2_ppm_delta2	Delta zum CO2-Wert vor 4 Minuten
co2_ppm_delta3	Delta zum CO2-Wert vor 6 Minuten
co2_ppm_delta4	Delta zum CO2-Wert vor 8 Minuten
co2_ppm_delta5	Delta zum CO2-Wert vor 10 Minuten
co2_ppm_delta6	Delta zum CO2-Wert vor 12 Minuten
co2_ppm_delta7	Delta zum CO2-Wert vor 14 Minuten
co2_ppm_delta8	Delta zum CO2-Wert vor 16 Minuten

Aus diesen Betrachtungen ergab sich folgender FV der für den weiteren Verlauf des Projektes genutzt wurde:

5.2. Ergebnisse

Da nicht alle Algorithmen auf die gleiche Weise evaluiert und grafisch dargestellt werden können, sollen die Ergebnisse zu

- Clustering Modellen
- Decision Tree Modellen
- Neuralen Netzwerken

einzeln betrachtet.

5.2.1. Decision Tree Modelle

Die ausgewählten Algorithmen konnten mit dem ausgewählten FV bei der Erwartungsberechnung über das gesamte Datenset eine Genauigkeit von etwa 94% erreichen. Eine *Confusion Matrix*, welche die errechneten Werte den tatsächlichen Labelwerten gegenüberstellt, zeigt, dass im hier benutzten Beispiel, ein Random Forest Classifier selbst bei der starken Unausgeglichenheit des Datensets ähnlich viele Fehler in sowohl An- als auch Abwesenheit macht.

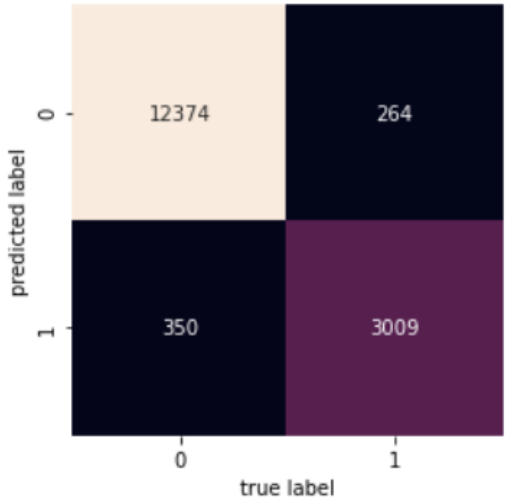

Abbildung 5.4.: Confusion Matrix eines RFC

In den Feldern oben links und unten rechts ist jeweils zu sehen, wann das Modell eine richtige Erwartung für jeweils Ab- und Anwesenheit errechnet hat, während die Felder oben rechts und unten links jeweils die falschen Erwartung für beide Werte zeigen.

Es ist ebenfalls aufschlussreich die erwarteten Labelwerte mit den tatsächlichen Labelwerten grafisch gegenüberzustellen, indem man das ursprüngliche Datenset mit timestamp und CO2-Wert zeichnet. Dabei wird die gemessene Anwesenheit farblich als gelb für an- und blau für abwesend eingefärbt.

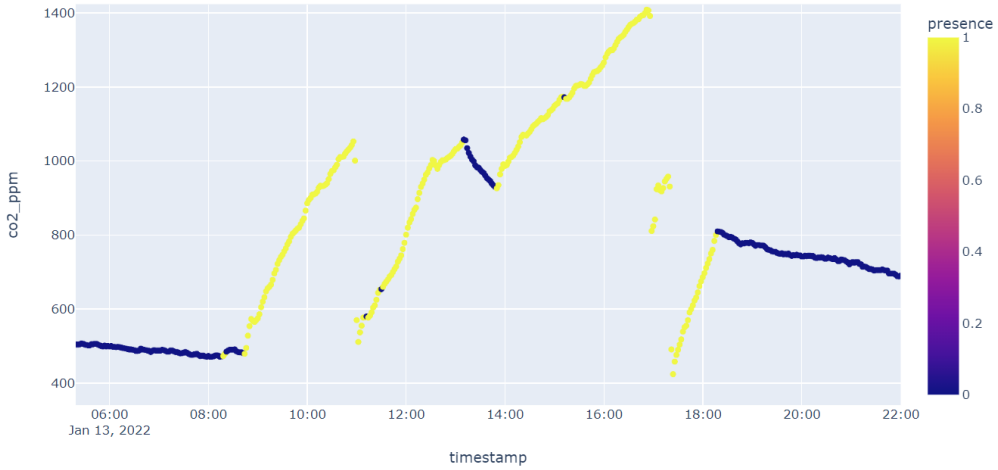

Abbildung 5.5.: Messwerte des 13. Januar

Löscht man aus dieser Datenreihe die tatsächlichen Labelwerte und fügt stattdessen die errechneten Anwesenheitswerte des Modells ein, zeigt sich die hohe Genauigkeit deutlich. Bis auf einige kleine Fehler trifft das Modell eine sehr präzise Aussage über die aktuelle Anwesenheit.

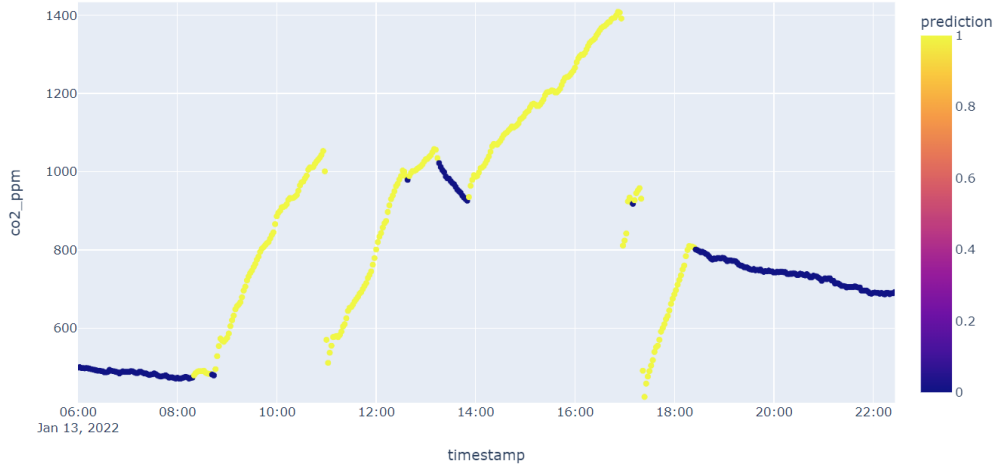

Abbildung 5.6.: Messwerte des 13. Januar

Dieses Ergebnis konnte über eine Vielzahl von Kombinationen verschiedener Räume und Modelle weiterhin bestätigt und repliziert werden. Die Tatsache, dass die Genauigkeiten aller Modelle so nah beieinander liegen, rührt daher, dass dieses Datenset ein typisches Klassifizierungsproblem darstellt, wodurch die sich wiederholenden Schemata in den CO2-Werten von einer Vielzahl von Algorithmen schnell erkannt und verarbeitet werden können.

Beim Tuning der ausgewählten Modelle wurden als Parameter-Optionen ausschließlich Werte gewählt, die nicht den Standardwerten der Modelle entsprechen. Somit sollte das GridSearchCV-Modul ausschließlich in der Menge von Parametern nach Möglichkeiten suchen, die nicht den Standard-Einstellungen entsprechen.

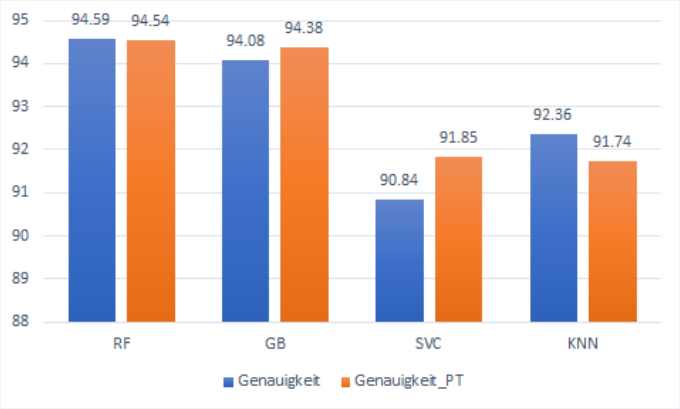

Abbildung 5.7.: !!PLACEHOLDER-GRAFIK!! Ergebnisse des Parameter Tuning

Es ist zu sehen, dass das Parameter Tuning die Genauigkeit der Modelle im erwarteten Rahmen erhöht oder verringert hat. Dies bestätigt die Annahme, dass dieser Anwendungsfall ein typisches Klassifizierungsproblem darstellt, weshalb die Modelle nicht mehr maßgeblich verbessert werden können. Der Schritt des Parameter Tunings sollte trotzdem grundsätzlich immer durchgeführt werden, um zu erkennen, ob man durch eine einfache Änderung der Parameter eine Verbesserung des Modells hervorbringen kann.

Zusätzlich wurden die *ROC-Curves* der True- und False-Positives gegeneinander gezeichent um zu erknennen inwiefern die Ergebnisse zufällig oder errechnet sind.

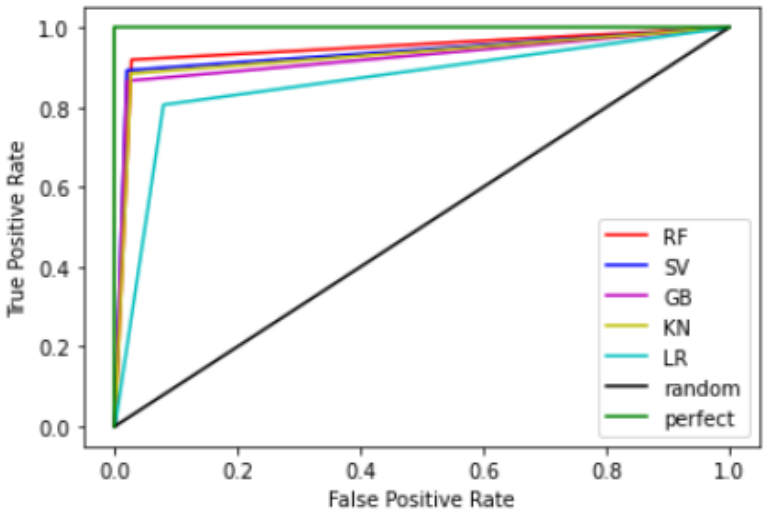

Abbildung 5.8.: ROC-Curves aller Modelle

Man sieht deutlich, wie alle Modelle sehr ähnliche Ergebnisse aufweisen. Keines der Modelle weist Anzeichen auf, dass Ergebnisse wegen Underfitting erraten werden.

5.2.2. Umgang mit fehlenden Präsenz-Labels

Da zur Auswertung auch eine große Menge an Datensätzen ohne Präsenz-Label zur Verfügung standen, wurden die trainierten Modelle zusätzlich auf diese Datensets angewandt und die Ergebnisse so gut wie möglich ausgewertet. Die Ergebnisse sind hier aufgrund des fehlenden Labels lediglich augenscheinlich zu bewerten, da eine mathematische Ermittlung der Genauikgeit nicht möglich ist.
Die Frage, die sich in diesem Kontext stellt, ist die, ob ein auf einen Büroraum trainiertes Modell auch richtige Erwartungen für einen Wohnraum treffen kann. Da das Modell unter anderen auf eine Erkennung der aktuellen Tageszeit trainert wird, muss geprüft werden, inwiefern ein solches Modell auch Aussagen über CO2-Werte treffen kann, dessen Veränderungen außerhalb der bisher trainierten Arbeitszeiten von etwa 7 Uhr morgens bis 17 Uhr Nachmittags befinden. Außerdem ist der Wohnraum deutlich größer, als ein Büro und hat in diesem Fall eine größere Anzahl anwesender Personen, wodurch sich die ermittelten Deltas der CO2-Werte erheblich von den Büroräumen unterscheiden.
Eines der Datensets zeichnete kontinuierliche Werte aus einem Wohnzimmer auf, in dem sich über einen Tag hinweg maximal drei Personen befanden. Die Zeiten, zu denen in diesem Wohnzimmer anhand der CO2-Werte Anwesenheit zu erkennen war, unterschied sich deutlich von den Datensets der Büroräume der FH-Aachen. Die starken Anstiege des CO2-Wertes waren bei diesem Datenset viel mehr über den Tag verteilt, wodurch sich die Datenreihe für diesen Test besonders anbot.

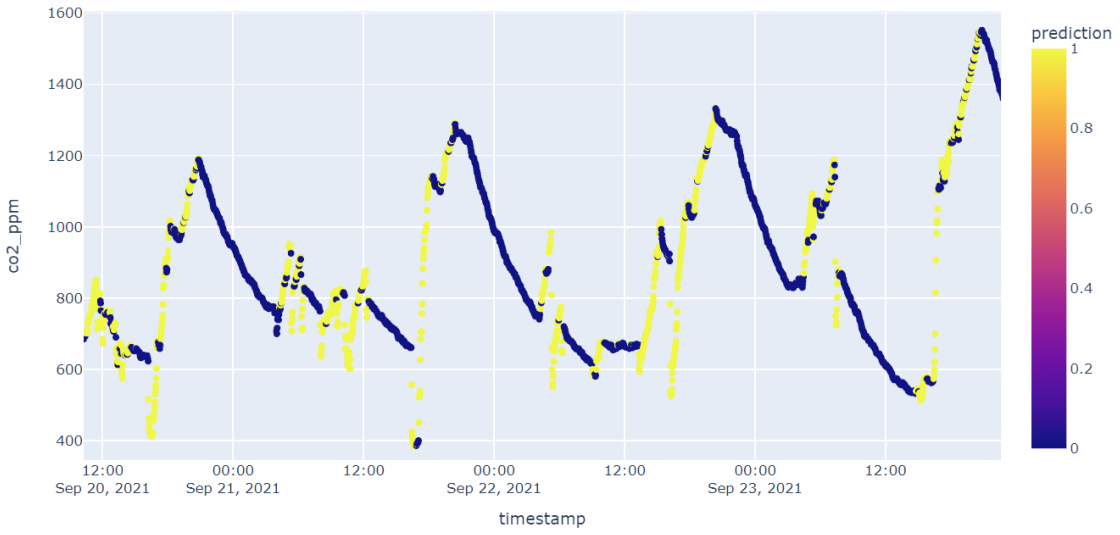

Abbildung 5.9.: Büroraum-Modell trifft Erwartungen für Wohnzimmer

Wie anhand der Einfärbung zu erkennen, ist, kann das Modell auch unabhängig von der Tageszeit und den bekannten Gegebenheiten eines normalen Arbeitstages Anwesenheit erkennen. Vorallem deutliche Anstiege und Abfälle werden deutlich jeweils Geld und Blau eingefärbt, während es auf einigen Strecken zwischen den Maxima und Minima nur gelegentlich zu Fehleinschätzungen kommt.

5.2.3. Neurale Netzwerke

Die beiden implementierten Neuronalen Netze konnten eine ähnliche Leistung, wie die bereits oben genannten Algorithmen erzielen. Beide Modelle wurden über so viele Iterationen (Epochen) trainiert, bis ersichtlich war, dass die Genauigkeit gegen einen Wert konvergiert. Zur Auswertung wurden hier die *Accuracy*- und *Loss*-Funktionen genutzt. Die Genauigkeit beschreibt, wie bei den anderen Modellen das Verhältnis zwischen richtigen und falschen Berechnungen.
Die Loss-Funktion hingegen gibt die Abweichung einer Schätzung zum tatsächlichen Wert an. Diese beiden Wert werden nach jeder Epoche ausgewertet, während zunächst der Erfolg während des Trainigs als *train* angegeben wird und danach das Modell an einem zufälligen Teil des Validierungssets *val* getestet wird. Liegen diese beiden Kurven nah beieinander, ist dies ein gutes Anzeichen für ein funktionierndes Modell ohne Over- oder Underfitting, da sowohl beim Training als auch bei der Validierung die Genauigkeiten Ähnlichkeiten aufweisen, während zugleich die Loss-Funktion minimiert wird.

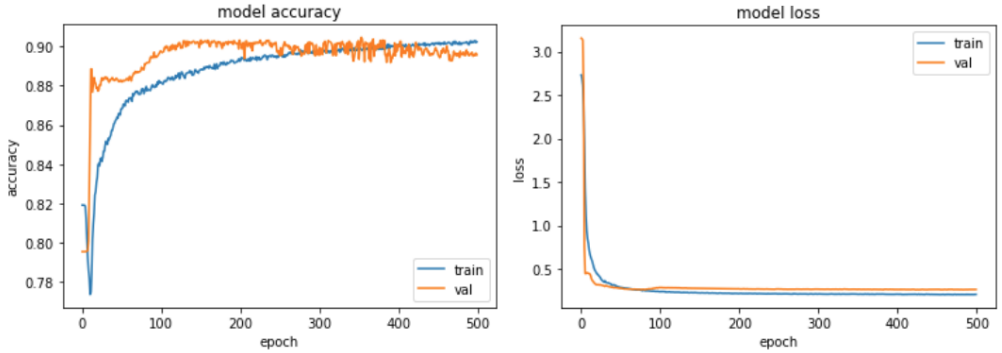

Abbildung 5.10.: Ergebnisse des NN

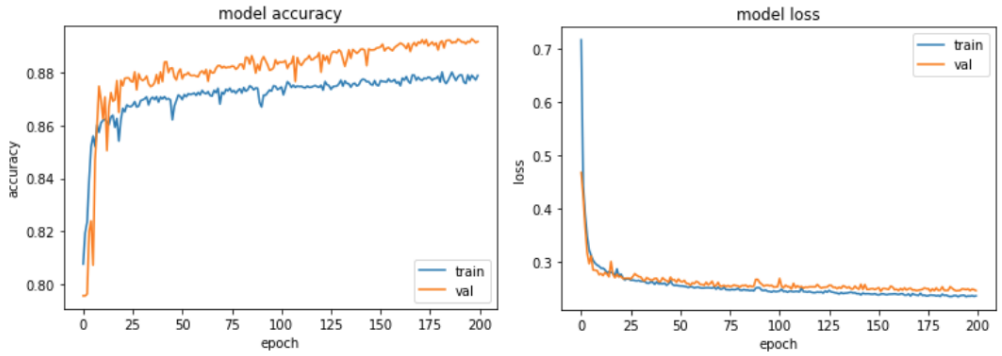

Abbildung 5.11.: Ergebnisse des LSTM

Wie zu sehen ist, weisen beide Arten neuraler Netzwerke diese Eigenschaften auf. Die Genauigkeiten beider Modelle konvergierten auch nach verschiedenen Epochen-Werten etwa bei 89%.

Tabelle 5.3.: Genauigkeitsauswertung

Präsenz	Precision	Recall	F1
0	0.94	0.79	0.86
1	0.50	0.82	0.62

5.2.4. Clustering Modelle

Clustering Modelle hatten von allen benutzten Algorithmen die größten Schwierigkeiten bei der Klassifikation des Datensets. Während die Genauigkeit bei der Erwartungsberechnung von Abwesenheit bei 94% lag, konnte das K-Means-Modell Anwesenheiten nur zu 50% erkennen, was bedeutet, dass das Modell das Ergebnis errät, anstatt ihn zuverlässig zu berechnen.
Wegen der Unausgeglichenheit des Datensets sind ca. zwei Drittel des Datensets sehr einfach zu klassifizieren, da es im ganzen Datenset nie Anwesenheiten zwischen beispielsweise 20:00 Uhr und 06:00 Uhr gibt. Selbst wenn das Modell beim letzten Drittel, also den Anwesenheiten rät und somit nur zu 50% trifft, suggeriert eine daraus resultierende Genauigkeit von etwa 80% ein funktionierendes Modell, allerdings zeigt sich hier, warum auch die Gegenüberstellung von True und False Positives in der Modellauswertung von entscheidender Wichtigkeit sein kann.

Wie anhand der Precision-Score zu sehen ist, tritt hier genau der oben beschriebene Fall ein.
Zur Veranschaulichung der Ergebnisse hilft eine Dimensionalitätsreduktion, bei der man das Datenset auf zwei Dimensionen reduziert und die An- und Abwesenheiten in einem Graphen darstellt.
Die Menge der Anwesenheiten (Orange) ist nahezu identisch mit der Teilmenge der Abwesenheiten (Blau). Das heißt, dass alle Datenpunkte die in der linken Hälfte des blauen Datensets liegen, sehr einfach klassifiziert werden können.

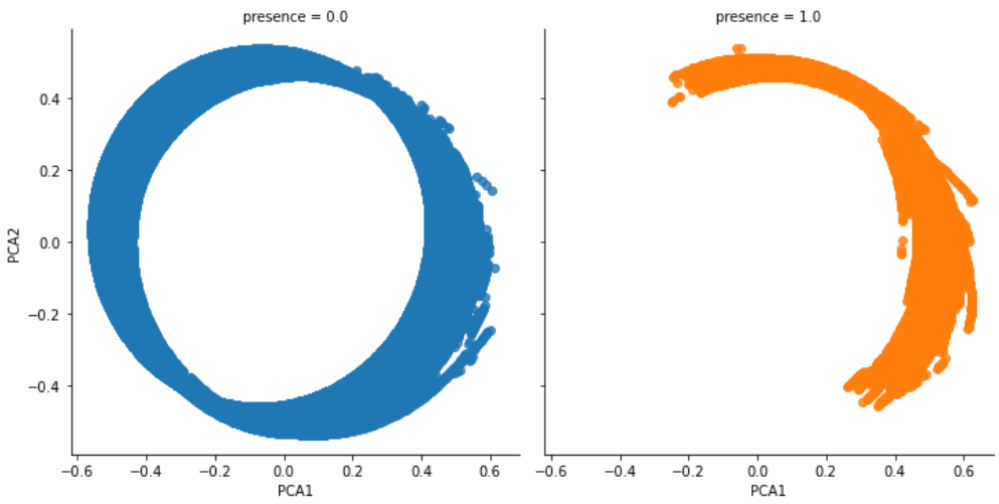

Abbildung 5.12.: Reduktion des Datensets auf zwei Dimensionen

Legt man beide Datensets übereinander ist es nun nahezu unmöglich zwischen An- und Abwesenheits zu unterscheiden.

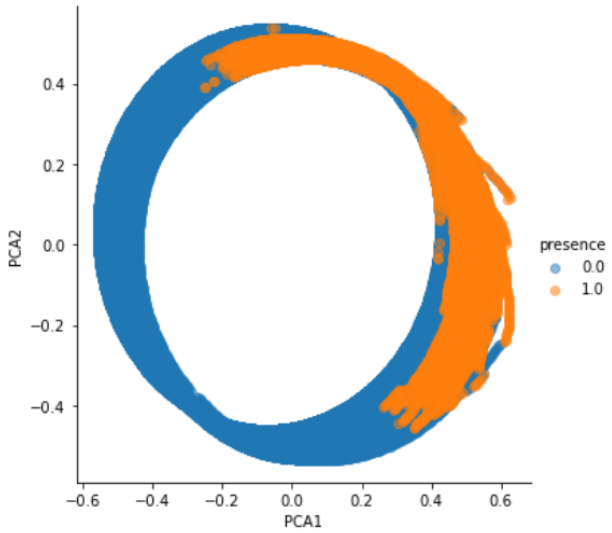

Abbildung 5.13.: Datenreduktion ohne Trennung

Das Erkennen von Abwesenheit im linken Teilbild ist nun erheblich einfacher, als die Trennung zwischen An- und Abwesenheit im rechten Teilbild.

6. Zusammenfassung und Ausblick

6.1. CO2 als Anwesenheitsindikator

Die Genauigkeit aller Modelle bei der Präsenzerkennung während der gesamten Durchfürungsphase des Projektes lag zwischen 88% bei neuronalen Netzwerken und 95% bei den Klassifizierungsalgorithmen. Die Beziehung zwischen der aktuellen Tageszeit und dem CO2-Gehalt der Luft, konnte erfolgreich bestätigt werden. Dieses Ergebnis wird ebenfalls gestützt von der vorangegangenen Forschung[1]. .
Weiterhin konnte gezeigt werden, dass alle Modelle zusätzlich in der Lage sind, menschliche Präsenz außerhalb der trainierten Uhrzeiten korrekt zu erkennen, wenn die Modelle mit Deltas zu Vergangenheitswerten des CO2-Gehalts trainiert werden. Die Präsenzerkennung ist also vollständig unabhängig von den Umständen des Trainigssets und kann sowohl für andere Räume, als auch zu anderen Tageszeiten effektiv genutzt werden. CO2 ist somit als geeigneter Indikator für menschliche Präsenz in Innenräumen anzusehen.

Die Zweckmäßigkeit von CO2-Sensoren für Gebäudeautomatisierungssysteme konnte als kostengünstige und leicht zu implemtentierende Option gezeigt werden.

6.2. Mögliche Verbesserungen

Die Genauigkeit von Machine Learning Modellen kann allgemein durch die Hinzugabe von mehr Daten in ein Modell verbessert werden. Da der Datenbestand, auf die sich diese Arbeit stützt, über die Bearbeitungszeit des Projektes kontinuierlich vergrößert wurde, wuchs damit auch die Anzahl an Anhaltspunkten für Zusammenhänge zwischen den Datenfeldern stetig an. Besonders nützlich wäre das Sammeln von Messdaten über ein oder mehrere Jahre hinweg. Somit hätten die Modelle eine Chance Gewohnheiten in z.B. Urlaubstagen, Feiertage oder auch wiederkehrende Meetings korrekt einzuordnen, sodass Gebäudeautomatisierungssysteme ein Maximum an Komfort und Energieeffizienz herstellen können.

Da Infrarotsensoren inhärent nicht in der Lage sind kontinuierliche Präsenz zu erkennen, wenn sich die Personen im Raum nicht bewegen, kam es innerhalb des Datensets immer wieder zu kleinen Messfehlern, die durch die Gruppierung und Durchschnittsberechnung der Präsenzwerte behoben werden mussten. Es ist klar davon auszugehen, dass eine genauere Datenlage auch die Qualität der trainierten Modelle steigern würde.
Zusätzlich existierten im Datenset durch seltene Probleme mit der Hardware längere Datenreihen mit falschen Labelwerten, welche während der Bearbeitung ausgeschlossen werden mussten. Da sich dadurch die für das Training geeignete Datenmenge verringerte, ist auch hier davon auszugehen, dass das Aussortieren dieser Werte mit einer Verringerung der Modellqualität einherging.

[1] [Hanfstaengl et al., 2019]

Quellenverzeichnis

[Hanfstaengl et al., 2019] Hanfstaengl, L., Parzinger, M., Wirnsberger, M., Spindler, U., and Wellisch, U. (2019). Identifying the presence of people in a room based on machine learning techniques using data of room control systems. Conference.

[Umweltbundesamt, 2020] Umweltbundesamt (2020). Endenergieverbrauch der privaten haushalte. Online. https://www.umweltbundesamt.de/daten/private-haushalte-konsum/wohnen/energieverbrauch-privater-haushalte/.

[Umweltbundesamt, 2022] Umweltbundesamt (2022). Energieverbrauch nach energieträgern und sektoren. Online. https://www.umweltbundesamt.de/daten/energie/energieverbrauch-nach-energietraegern-sektoren.

Abkürzungsverzeichnis

g	Gravitation in Nähe der Erdoberfläche
Nu	Nußelt-Zahl
ν_{Luft}	Kinematische Viskosität von Luft
Pr	Prandtl-Zahl
$\dot{Q}$	Wärmestrom
Ra	Rayleigh-Zahl
ρ_{Luft}	Dichte von Luft
T	Temperatur
T_∞	Umgebungstemperatur

Abbildungsverzeichnis

Tabellenverzeichnis

A. Quellcode

```
# Drill-Funktionen importieren
import drill as drill
import modelTuning as modelTuning
import prepareData as prepareData
import validation as validation
import createModel as createModel

import requests
import numpy as np
import json
import pandas as pd
import seaborn as sns
import datetime as dt
from math import sqrt

import matplotlib.pyplot as mplt
import plotly.express as plt
import plotly.graph_objects as go
from plotly.subplots import make_subplots

from sklearn.model_selection import train_test_split,
    RandomizedSearchCV, GridSearchCV
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix,
    mean_squared_error, classification_report
from sklearn.mixture import GaussianMixture
from sklearn.datasets import make_blobs
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier,
    GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.datasets import load_digits
from sklearn import metrics
from sklearn.metrics import roc_curve
from sklearn import preprocessing
from sklearn.decomposition import PCA
from sklearn.manifold import Isomap

# Clustering
from sklearn.pipeline import Pipeline
from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons
from sklearn.metrics import adjusted_rand_score
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

import tensorflow as tf
from keras.models import Sequential
```

```
49 from keras.layers import Dense
50 from keras.layers import LSTM
51 from keras.layers import Dropout
52 from keras.layers import SpatialDropout1D
53
54 pd.options.mode.chained_assignment = None  # default='warn'
55
56 #rooms = {'0':'Daniel', '1':'Felix N#2','2':'Calvin','3':'bigDataLab',
57 #         '4':'FelixAkku', '7':'Lukasbuero','9':'Felix B. #1','10':'
       Felix N #1'}
58
59 #for room in rooms:
60 #    print('Processing ' + rooms[room])
61 #    df = pd.DataFrame
62 #    df = drill.get_PIR_data(room=room, presence=True)
63 #    if not df.empty:
64 #        df.to_json('data\\' + room + '.json')
65
66 #df = drill.get_PIR_data(room='H215', presence=True)
67 #df.to_json('data\H215.json')
68 #df = drill.get_PIR_data(room='H216', presence=True)
69 #df.to_json('data\H216.json')
70 #df = drill.get_PIR_data(room='H217', presence=True)
71 #df.to_json('data\H217.json')
72 #df = drill.get_PIR_data('dfs', False)
73 #df.to_json('data\test.json')
74
75 df = pd.read_json('data\\H217.json')
76 df_comp = pd.read_json('data\\living_room.json')
77
78 #plt.scatter(df, x='timestamp', y='co2_ppm', color='presence')
79
80 df_new = prepareData.preProcessDataset(df)
81 df_comp = prepareData.preProcessDataset(df_comp)
82
83 #df_new.head()
84
85 plt.scatter(df_new, x='timestamp', y='co2_ppm', color='presence')
86
87 # timestamp, presence, temperatur und humidity entfernen
88 # temp/humid erhoehen Genauigkeit deutlich, da relativ unverlaesslich
       -> von zu vielen aeusseren Faktoren abhaengig
89 df_timestamp = df_new['timestamp']
90 y_presence = df_new['presence']
91 #, 'temperature_celsius', 'relative_humidity_percent'
92 X_presence = df_new.drop(['timestamp', 'second', 'presence', '
       temperature_celsius', 'relative_humidity_percent',
93                          'upload_date', 'measurement_count', 'battery'
       ], axis=1)
94 X_presence_scaled, scaler = prepareData.normalize_min_max(X_presence)
95 #X_presence = X_presence.drop(['second_sin', 'second_cos'], axis=1)
96
```

```
97  Xtrain, Xtest, ytrain, ytest = train_test_split(X_presence, y_presence,
        test_size=0.2, random_state=1, shuffle=False)
98  Xtrain_scaled, Xtest_scaled, ytrain_scaled, ytest_scaled =
        train_test_split(X_presence_scaled, y_presence, test_size=0.2,
        random_state=1, shuffle=False)
99
100 # shift des trainings-sets um n minuten in die Vergangenheit
101 # -> Test ob Model auch in die Zukunft Erwartungen treffen kann
102 #ytrain = ytrain.shift(-5)
103 #ytrain = ytrain.replace(np.nan, 0)
104
105 # Create Classifiers and save to disk
106 classifiers = ['RFC', 'SVC', 'GBC', 'KNN', 'LR']
107
108 #for x in classifiers:
109 #     if (x == 'LR'):
110 #         createModel.createClassifier(x, Xtrain_scaled, ytrain_scaled)
111 #     else:
112 #         createModel.createClassifier(x, Xtrain, ytrain)
113
114 accuracies = {}
115 roc_curves = {}
116 for x in classifiers:
117     model = modelTuning.loadModel('models\\' + x + '.mod')
118     if (x == 'LR'):
119         ypred = model.predict(Xtest_scaled)
120         roc_curves[x] = roc_curve(ytest_scaled, ypred)
121     else:
122         ypred = model.predict(Xtest)
123         roc_curves[x] = roc_curve(ytest, ypred)
124     #rf_base_accuracy = modelTuning.evaluateClassifier(model,
        X_presence, y_presence)
125     #accuracies[x] = rf_base_accuracy
126 #df_acc = pd.DataFrame(data=accuracies, dtype=float, index=['0'])
127
128 createModel.createClassifier('RFC', Xtrain, ytrain)
129
130 # Verschiedene Feature Vektoren vergleichen
131 #df1 = pd.read_json('validation\\d15_noShift.json')
132 #df2 = pd.read_json('validation\\d15.json')
133 #df3 = pd.read_json('validation\\temp_hum_d15.json')
134 #frames = [df1, df2, df3]
135 #result = pd.concat(frames)
136
137 mplt.plot(roc_curves['RFC'][0], roc_curves['RFC'][1],'r-',label = 'RF')
138 mplt.plot(roc_curves['SVC'][0], roc_curves['SVC'][1],'b-',label = 'SV')
139 mplt.plot(roc_curves['GBC'][0], roc_curves['GBC'][1],'m-',label = 'GB')
140 mplt.plot(roc_curves['KNN'][0], roc_curves['KNN'][1],'y-',label = 'KN')
141 mplt.plot(roc_curves['LR'][0], roc_curves['LR'][1],'c-',label = 'LR')
142 #mplt.plot(fpr_LR,tpr_LR,'b-', label= 'LR')
143 mplt.plot([0,1],[0,1],'k-',label='random')
144 mplt.plot([0,0,1,1],[0,1,1,1],'g-',label='perfect')
145 mplt.legend()
```

```
146 mplt.xlabel('False Positive Rate')
147 mplt.ylabel('True Positive Rate')
148 mplt.show()
149
150 model = modelTuning.loadModel('models\\' + 'RFC' + '.mod')
151 ypred = model.predict(Xtest)
152 rf_base_accuracy = modelTuning.evaluateClassifier(model, X_presence,
        y_presence)
153 print(rf_base_accuracy)
154
155 mat = confusion_matrix(ytest, ypred)
156 sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)
157 mplt.xlabel('true label')
158 mplt.ylabel('predicted label')
159
160 print(classification_report(ytest, ypred))
161
162 df_valid_class = Xtest.copy()
163 df_valid_class['timestamp'] = df_timestamp
164 df_valid_class['prediction'] = ypred
165 df_valid_class['co2_ppm'] = df_new['co2_ppm']
166
167 plt.scatter(df_valid_class, x='timestamp', y='co2_ppm', color='
        prediction')
168
169 df_comp
170
171 # Trainiertes Modell auf Wohnzimmer anwenden und plotten
172 df_timestamp = df_comp['timestamp']
173 df_comp = df_comp.drop(['timestamp', 'second'], axis=1)
174
175 model = modelTuning.loadModel('models\RFC.mod')
176 ypred = model.predict(df_comp)
177 df_valid_class = df_comp.copy()
178 df_valid_class['timestamp'] = df_timestamp
179 df_valid_class['prediction'] = ypred
180 df_valid_class['co2_ppm'] = df_comp['co2_ppm']
181
182 plt.scatter(df_valid_class, x='timestamp', y='co2_ppm', color='
        prediction')
183
184 df_plot = pd.Series(model.feature_importances_, index=Xtrain.columns).
        sort_values()
185 df_plot.plot(kind='barh', legend=False, width=0.8, figsize=(7,7),
        sort_columns=True)
186 #plt.show()
187
188 # plot feature-importance
189 if (model.feature_importances_.any):
190     feature_imp = pd.Series(model.feature_importances_,index=Xtrain.
        columns).sort_values(ascending=False)
191     # Creating a bar plot
192     ax = sns.barplot(x=feature_imp, y=feature_imp.index)
```

```
193     ax.set(xlabel='Importance', ylabel='Feature')
194
195 from mpl_toolkits import mplot3d
196
197 modelPCA = PCA(n_components=3)
198 modelPCA.fit(X_presence_scaled)
199 X_2D = modelPCA.transform(X_presence_scaled)
200
201 df_new['PCA1'] = X_2D[:, 0]
202 df_new['PCA2'] = X_2D[:, 1]
203 df_new['PCA3'] = X_2D[:, 2]
204 sns.lmplot(x="PCA1", y="PCA2", hue='presence', data=df_new, fit_reg=
        False, scatter_kws={'alpha':0.5})
205
206 fig = mplt.figure(figsize=[16.4, 14.8])
207 ax = mplt.axes(projection='3d')
208 ax.scatter3D(df_new['PCA1'], df_new['PCA2'], df_new['PCA3'], c=df_new['
        presence']);
209
210 df1 = df_new.head(8000)
211
212 df_clus = pd.DataFrame()
213 df_clus['presence'] = df1['presence']
214 df_clus['second_sin'] = df1['second_sin']
215 df_clus['second_cos'] = df1['second_cos']
216 #df_clus['co2_ppm'] = df1['co2_ppm']
217 df_clus['co2_ppm_delta1'] = df1['co2_ppm_delta1']
218 df_clus['co2_ppm_delta5'] = df1['co2_ppm_delta5']
219 df_clus['dayofweek'] = df1['dayOfWeek']
220
221 iso = Isomap(n_components=2)
222 iso.fit(df_clus)
223 data_projected = iso.transform(df_clus)
224
225 plt.scatter(data_projected, x=data_projected[:, 0], y=data_projected[:,
        1], color=df_clus['presence'], opacity=0.85,
226             labels={
227              "x": "",
228              "y": "",
229         },)
230
231 from sklearn.naive_bayes import GaussianNB
232 model = GaussianNB()
233 model.fit(Xtrain, ytrain)
234 y_model = model.predict(Xtest)
235 print(accuracy_score(ytest, y_model))
236
237 print(classification_report(ytest, y_model))
238
239 df_valid_class = Xtest.copy()
240 df_valid_class['timestamp'] = df_timestamp
241 df_valid_class['prediction'] = y_model
242 df_valid_class['co2_ppm'] = df_new['co2_ppm']
```

```
243
244 plt.scatter(df_valid_class, x='timestamp', y='co2_ppm', color='
        prediction')
245
246 # unsupervised learning: Clustering
247 modelKM = GaussianMixture(n_components=2, covariance_type='full')
248 modelKM.fit(Xtrain)
249 y_ggm = modelKM.predict(Xtest)
250 print(accuracy_score(ytest, y_model))
251
252 print(classification_report(ytest, y_ggm))
253
254 y_ggm = np.logical_not(y_ggm).astype(int)
255
256 df_valid_ggm = Xtest.copy()
257 df_valid_ggm['timestamp'] = df_timestamp
258 df_valid_ggm['prediction'] = y_ggm
259 df_valid_ggm['co2_ppm'] = df_new['co2_ppm']
260
261 plt.scatter(df_valid_ggm, x='timestamp', y='co2_ppm', color='prediction
        ')
262 #df_new['gaussian_mixture'] = y_ggm
263 #plt.scatter(df_new, x='PCA1', y="PCA2", color='gaussian_mixture',
        opacity=0.5)
264
265 kmeans = KMeans(n_clusters=2)
266 kmeans.fit(Xtrain)
267 y_km = kmeans.predict(Xtest)
268 #y_km = np.logical_not(y_km).astype(int)
269 print(accuracy_score(ytest, y_km))
270
271 print(classification_report(ytest, y_km))
272
273 #df_new['k_means'] = y_km
274 #plt.scatter(df_new, x='timestamp', y="co2_ppm", color='k_means')
275
276 df_valid_km = Xtest.copy()
277 df_valid_km['timestamp'] = df_timestamp
278 df_valid_km['prediction'] = y_km
279 df_valid_km['co2_ppm'] = df_new['co2_ppm']
280
281 plt.scatter(df_valid_km, x='timestamp', y='co2_ppm', color='prediction'
        )
282
283 #modelTuning.parameterTuning('RF', Xtrain, ytrain, Xtest, ytest,
        X_presence, y_presence)
284 #modelTuning.parameterTuning('GB', Xtrain, ytrain, Xtest, ytest,
        X_presence, y_presence)
285 #modelTuning.parameterTuning('SVC', Xtrain, ytrain, Xtest, ytest,
        X_presence, y_presence)
286 #modelTuning.parameterTuning('KNN', Xtrain, ytrain, Xtest, ytest,
        X_presence, y_presence)
287
```

```
288 X_presence, scaler = prepareData.normalize_min_max(X_presence)
289 #Xtest, _ = normalize_min_max(Xtrain, scaler)
290
291 timesteps = 25
292 X, y = prepareData.reshape_data_for_LSTM(X_presence, y_presence,
        timesteps)
293
294 Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2,
        random_state=1, shuffle=False)
295
296 model = createModel.createLSTM(Xtrain)
297 model.summary()
298
299 history_lstm = model.fit(Xtrain, ytrain, epochs=200, batch_size=8,
        verbose=2, validation_data=(Xtest, ytest))
300
301 model = createModel.createNN(Xtrain)
302 #model.summary()
303
304 history = model.fit(X_presence, y_presence, epochs=100,
        validation_split=0.2)
305
306 #val_loss, val_acc = model.evaluate(X, y)
307 #print(val_loss, val_acc)
308
309 mplt = validation.plotHistoryAccuracy(history_lstm)
310 mplt.show()
311
312 mplt = validation.plotHistoryLoss(history_lstm)
313 mplt.show()
314
315 mplt = validation.plotHistoryAccuracy(history)
316 mplt.show()
317
318 mplt = validation.plotHistoryLoss(history)
319 mplt.show()
```

app/Anwesenheitsanalyse.py

```
1  import modelTuning as modelTuning
2  import prepareData as prepareData
3
4  from sklearn.ensemble import BaggingClassifier,
       GradientBoostingClassifier, RandomForestClassifier
5  from sklearn.linear_model import LogisticRegression
6  from sklearn.svm import SVC
7  from sklearn.neighbors import KNeighborsClassifier
8
9  import tensorflow as tf
10 from keras.models import Sequential
11 from keras.layers import Dense
12 from keras.layers import LSTM
13 from keras.layers import Dropout
```

```
14 from keras.layers import SpatialDropout1D
15 from keras import optimizers
16
17 import numpy as np
18
19 def createClassifier(classType, Xtrain, ytrain):
20     if (classType == 'RFC'):
21         modelClass = RandomForestClassifier()
22     elif (classType == 'SVC'):
23         modelClass = SVC()
24     elif (classType == 'GBC'):
25         modelClass = GradientBoostingClassifier()
26     elif (classType == 'KNN'):
27         modelClass = KNeighborsClassifier()
28     elif (classType == 'LR'):
29         modelClass = LogisticRegression(solver='saga', max_iter=5000)
30         #Xtrain, scaler = prepareData.normalize_min_max(Xtrain)
31         #ytrain, scaler = prepareData.normalize_min_max(ytrain)
32
33     modelClass.fit(Xtrain, ytrain)
34     modelTuning.saveModel(modelClass, 'models\\' + classType + '.mod')
35
36 def createLSTM(Xtrain):
37     model = Sequential()
38     model.add(LSTM(units=5, return_sequences=True, input_shape = (
       Xtrain.shape[1],Xtrain.shape[2]), activation='sigmoid'))
39     model.add(Dropout(0.2))
40 #     model.add(LSTM(units=1, return_sequences=True, input_shape = (
       Xtrain.shape[1],Xtrain.shape[2]), activation='relu'))
41 #     model.add(Dropout(0.2))
42     model.add(Dense(1, activation='relu'))
43
44     opt = tf.keras.optimizers.Adam(learning_rate=1e-3, decay=1e-5)
45     model.compile(loss='binary_crossentropy', optimizer=opt, metrics='
       accuracy')
46     #model.compile(loss='binary_crossentropy', optimizer='rmsprop',
       metrics='accuracy')
47     #sgd = tf.keras.optimizers.SGD(learning_rate=0.01, decay=1e-4,
       momentum=0.7, nesterov=True)
48     #model.compile(loss='mean_squared_error', optimizer=sgd, metrics='
       accuracy')
49     model.build()
50
51     return model
52
53 def createNN(Xtrain):
54     model = tf.keras.models.Sequential()
55     model.add(tf.keras.layers.Flatten())
56     model.add(tf.keras.layers.Dense(Xtrain.shape[1], input_dim=Xtrain.
       shape[1], activation='relu'))#, activation=tf.nn.relu))
57     model.add(Dropout(0.2))
58     #model.add(tf.keras.layers.Dense(round(Xtrain.shape[1]/2),
       activation='relu'))#, activation=tf.nn.relu))
```

```
59      model.add(tf.keras.layers.Dense(round(Xtrain.shape[1]/3),
        activation='relu'))#, activation=tf.nn.relu))
60      model.add(tf.keras.layers.Dense(1, activation='relu'))
61
62      opt = tf.keras.optimizers.Adam(learning_rate=1e-5)#, decay=1e-5)
63      model.compile(optimizer=opt,
64                    loss='binary_crossentropy',
65                    metrics=['accuracy'])
66  #     model.build()
67
68      return model
```

app/createModel.py

```
1   import requests
2   import pandas as pd
3   import datetime as dt
4
5   def connect_drill(query, caching=True, chunk_size: int = 0):
6       #username = os.getenv("DRILLUSERNAME")
7       host = 'https://proxima.bigdata.fh-aachen.de:8047'
8       username = 'al7739s'
9       password = 'tWtx4UYhTdUbPHumX3VixMhdi'
10      headers = {'Content-Type': 'application/json',
11                 'Authorization': '%s:%s' % (username, password)}
12      #headers = {'Authorization': username + ':' + password}
13      if caching:
14          headers["Cache-Control"] = "max-age=" + "1440"
15      else:
16          headers["Cache-Control"] = "max-age=" + "0"
17      #if chunk_size > 0:
18          #headers["format"] = "chunks:" + str(chunk_size)
19      data = {'query': "{q}".format(q=query)}
20
21      try:
22          result = requests.post(host + '/query', json=data, headers=
        headers, verify=True)
23          print(result)
24      except Exception as e:
25          print("The drill-proxy is not reachable. Please check if you
        are in the FH-Aachen network.")
26          raise (e)
27
28      data = None
29      try:
30          data = pd.read_json(result.text)
31          if data.empty:
32              print('Result of query is empty!')
33              print('Query was: ' + query)
34      except ValueError:
35          print("Something went wrong when converting the json string
        from the datasource to a pandas DataFrame.")
36          print(result.text)
```

```
37      return data
38
39  #     query = """SELECT *
40  #                 FROM dfs.co2meter.`sensor_data`
41  #                 WHERE `serial_number` = 's_3c6105d3abae_299589'
42  #                 FETCH FIRST 100000 ROWS ONLY"""
43
44  #     query = """SELECT `timestamp`,`room`, `presence`, `co2_ppm`, `
        temperature_celsius`, `relative_humidity_percent`
45  #                 FROM ipenv.data.`sensor_data_v1`
46  #                 WHERE `timestamp` > 1627776000
47  #                 AND `room` LIKE '{room}'
48  #                 LIMIT 1000000""".format(room=room)
49
50  #                 WHERE SensorID LIKE 's_e8db84c5f33d_281913'
51
52  #{"room":{"0":"Daniel","1":"Felix N #2","2":"Calvin","3":"bigDataLab
        ","4":"FelixAkku","5":"Galla","6":null,"7":"Lukasbuero","8":"Remmy
        ","9":"Felix B. #1","10":"Felix N #1","11":"Elsen","12":"Internal
        Server Error"}}
53
54
55  def get_PIR_data(room: str = "H217", presence = True):
56      dict_rooms = {'dfs': 'dfs', 'H217': 'Elsen', 'H216': 'Galla', 'H215
        ': 'Remmy', '0':'Daniel',
57                    '1':'Felix N#2','2':'Calvin','3':'bigDataLab','4':'
        FelixAkku', '7':'Lukasbuero','9':'Felix B. #1','10':'Felix N #1'}
58      room = dict_rooms[room]
59
60      query = """SELECT *
61                  FROM ipenv.data.`sensor_data_v1`
62                  WHERE `timestamp` > 1627776000
63                  AND `room` LIKE '{room}' LIMIT 10000000""".format(room=
        room)
64  #                   LIMIT 1000000""".format(room=room)
65
66      pir_data = pd.DataFrame
67      pir_data = connect_drill(query, caching=True)
68
69      if (pir_data.empty):
70          return pir_data
71
72      pd.set_option("display.max_rows", None, "display.max_columns", None
        )
73
74      # apply CET time offset to timestamp
75      pir_data["timestamp"].dt.tz_localize('Europe/Berlin', ambiguous=
        True, nonexistent='shift_forward')
76      pir_data["timestamp"] = pir_data["timestamp"] + pd.Timedelta(hours
        =2)
77
78      if (presence):
79          pir_data["presence"] = pir_data["presence"].astype(int)
```

```
80
81      pir_data = pir_data.groupby(pd.Grouper(key="timestamp", freq="2min"
        )).mean()\
82          .round(0).reset_index(drop=False)
83
84      return pir_data
```

app/drill.py

```
1  import numpy as np
2  import pickle
3
4  from sklearn.ensemble import GradientBoostingClassifier,
       RandomForestClassifier
5  from sklearn.neighbors import KNeighborsClassifier
6  from sklearn.svm import SVC
7
8  from sklearn.metrics import accuracy_score, classification_report
9  from sklearn.model_selection import cross_val_score, GridSearchCV
10
11 def saveModel(model, filename):
12     pickle.dump(model, open(filename, 'wb'))
13     return
14
15 def loadModel(filename):
16     loaded_model = pickle.load(open(filename, 'rb'))
17     return loaded_model
18
19 def evaluateClassifier(model, test_features, test_labels):
20     accuracy = np.mean(cross_val_score(model, test_features,
       test_labels, cv=3))
21     return accuracy
22
23 def parameterTuning(modelType, Xtrain, ytrain, Xtest, ytest, X_presence
       , y_presence):
24     if (modelType == 'RF'):
25         classifier = RandomForestClassifier()
26         base_class = RandomForestClassifier()
27         param_test = {
28             'n_estimators':[50, 100, 150, 200],
29             'max_depth':[100, 125, 150],
30             'min_samples_split':[10, 25, 50, 75],
31             'min_samples_leaf':[1, 2, 4, 6, 8],
32             'max_features':[2, 3, 4, 5]
33             }
34     elif (modelType == 'GB'):
35         classifier = GradientBoostingClassifier()
36         base_class = GradientBoostingClassifier()
37         param_test = {
38             'n_estimators':[25, 50, 100, 250],
39             'max_depth': [2, 5, 10, 15],
40             'min_samples_split':[25, 150, 500],
41             'min_samples_leaf':[50, 75, 125],
```

```
42              'max_features':[5, 6, 7, 8, 9],
43              'subsample':[0.75, 0.9, 0.95]
44               }
45      elif (modelType == 'SVC'):
46          classifier = SVC()
47          base_class = SVC()
48          param_test = {
49              'kernel': ['rbf', 'sigmoid'],
50              'C': [50, 100, 250, 500, 750, 1000],
51              'gamma': ['scale', 'auto']
52          }
53      elif (modelType == 'KNN'):
54          classifier = KNeighborsClassifier()
55          base_class = KNeighborsClassifier()
56          param_test = {
57              'n_neighbors': [5, 10, 25, 50],
58              'weights': ['uniform', 'distance'],
59              'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
60              'leaf_size': [60, 90, 120, 150],
61              'p': [1, 2, 3, 4]
62          }
63
64      base_class.fit(Xtrain,ytrain)
65      base_accuracy = evaluateClassifier(base_class, X_presence,
        y_presence)
66      ypred = base_class.predict(Xtest)
67      print(classification_report(ytest, ypred))
68
69      gsearch1 = GridSearchCV(estimator = classifier,param_grid =
        param_test, scoring='average_precision', verbose=5, n_jobs=8, cv=5)
70      gsearch1.fit(Xtrain,ytrain)
71
72      print('\nOptimized Parameters of ' + modelType + ':')
73      print(gsearch1.best_params_)
74
75      random_accuracy = evaluateClassifier(gsearch1.best_estimator_,
        X_presence, y_presence)
76      ypred = gsearch1.best_estimator_.predict(Xtest)
77      print(classification_report(ytest, ypred))
78
79      print('\nComparison Results of ' + modelType + ':')
80      print('Base Accuracy: {:0.2f}%.'.format(100 * base_accuracy))
81      print('Optimized Accuracy: {:0.2f}%.'.format(100 * random_accuracy)
        )
82      print('Improvement of {:0.2f}%.'.format(100 * (random_accuracy -
        base_accuracy) / base_accuracy))
83
84      saveModel(gsearch1.best_estimator_, 'models\\' + modelType + '.mod'
        )
85
86  def crossValidation(model, X_presence, y_presence, cv):
87      print(cross_val_score(model, X_presence, y_presence, cv=cv))
```

app/modelTuning.py

```
1  import numpy as np
2  import pandas as pd
3  from sklearn.preprocessing import MinMaxScaler, StandardScaler
4
5  def encodeCyclical(df, col, max_val):
6      df[col + '_sin'] = np.sin(2 * np.pi * df[col]/max_val)
7      df[col + '_cos'] = np.cos(2 * np.pi * df[col]/max_val)
8      return df
9
10 def detectOutliers(df):
11     x = df['co2_ppm']
12     q1 = np.percentile(x, 5)
13     q3 = np.percentile(x, 95)
14     iqr = q3 - q1
15     floor = q1 - 1.5*iqr
16     ceiling = q3 + 1.5*iqr
17     outlier_indices = list(x.index[(x < floor) | (x > ceiling)])
18     outlier_values = list(x[outlier_indices])
19     print(outlier_values)
20     return outlier_indices
21
22 def extractSection(df, dateFrom, dateTo):
23     df_new = df.loc[(df['timestamp'] > pd.to_datetime(dateFrom, unit='s
       ', origin='unix')) &
24                     (df['timestamp'] < pd.to_datetime(dateTo, unit='s',
       origin='unix'))]
25     return df_new
26
27 def dropSection(df, dateFrom, dateTo):
28     df_new = df.drop(df[(df['timestamp'] > pd.to_datetime(dateFrom,unit
       ='s',origin='unix')) &
29                         (df['timestamp'] < pd.to_datetime(dateTo,unit='
       s',origin='unix'))].index)
30     return df_new
31
32 def convertTimestamp(df):
33     # timestamp etwas leichter zu verarbeiten, wenn als Integer
       gespeichert
34     df = df.assign(second=lambda d: (d['timestamp'].dt.hour.astype('int
       ') * 3600 +
35                                      d['timestamp'].dt.minute.astype('int
       ') * 60 +
36                                      d['timestamp'].dt.second.astype('int
       ')))
37     return df
38
39 def dropBetweenTimestamp(df, timeFrom, timeTo):
40     df_new = df.drop(df[(df['hour'] > timeFrom) |
41                         (df['hour'] < timeTo)].index, inplace=False)
42     return df_new
```

```
43
44 def addDelta(df, col, amount):
45     df[col + '_delta' + str(amount)] = df[col] - df.shift(amount)[col]
46     return df
47
48 def insertWeekday(df):
49     df['dayOfWeek'] = df['timestamp'].dt.dayofweek
50     return df
51
52 def preProcessDataset(df):
53     # Daten mit falschen PIR-Werten rausschmeissen
54     df_new = dropSection(df, 1634518800, 1637586000)
55
56     # Timestamp einfacher zu verarbeiten, wenn als Integer gespeichert
57     df_new = convertTimestamp(df_new)
58     # Integer Timestamp jetzt zyklisch encodieren
59     df_new = encodeCyclical(df_new, 'second', 86400)
60
61     # Deltas einfuegen
62     df_new = addDelta(df_new, 'co2_ppm', 1)
63     #df_new = addDelta(df_new, 'co2_ppm', 2)
64     df_new = addDelta(df_new, 'co2_ppm', 3)
65     #df_new = addDelta(df_new, 'co2_ppm', 4)
66     df_new = addDelta(df_new, 'co2_ppm', 5)
67     #df_new = addDelta(df_new, 'co2_ppm', 6)
68     #df_new = addDelta(df_new, 'co2_ppm', 7)
69     #df_new = addDelta(df_new, 'co2_ppm', 8)
70     #df_new = addDelta(df_new, 'co2_ppm', 9)
71     #df_new = addDelta(df_new, 'co2_ppm', 10)
72     #df_new = addDelta(df_new, 'co2_ppm', 11)
73     #df_new = addDelta(df_new, 'co2_ppm', 12)
74     #df_new = addDelta(df_new, 'co2_ppm', 13)
75     #df_new = addDelta(df_new, 'co2_ppm', 14)
76     #df_new = addDelta(df_new, 'co2_ppm', 15)
77
78 #     df_new = addDelta(df_new, 'temperature_celsius', 1)
79 #     df_new = addDelta(df_new, 'temperature_celsius', 2)
80 #     df_new = addDelta(df_new, 'temperature_celsius', 3)
81 #     df_new = addDelta(df_new, 'temperature_celsius', 4)
82 #     df_new = addDelta(df_new, 'temperature_celsius', 5)
83 #     df_new = addDelta(df_new, 'temperature_celsius', 6)
84 #     df_new = addDelta(df_new, 'temperature_celsius', 7)
85 #     df_new = addDelta(df_new, 'temperature_celsius', 8)
86
87 #     df_new = addDelta(df_new, 'relative_humidity_percent', 1)
88 #     df_new = addDelta(df_new, 'relative_humidity_percent', 2)
89 #     df_new = addDelta(df_new, 'relative_humidity_percent', 3)
90 #     df_new = addDelta(df_new, 'relative_humidity_percent', 4)
91 #     df_new = addDelta(df_new, 'relative_humidity_percent', 5)
92 #     df_new = addDelta(df_new, 'relative_humidity_percent', 6)
93 #     df_new = addDelta(df_new, 'relative_humidity_percent', 7)
94 #     df_new = addDelta(df_new, 'relative_humidity_percent', 8)
95
```

```
96      # Daten shiften
97      #df_new['co2_ppm_last'] = df_new.shift(1)['co2_ppm']
98
99      # Werte von vor n-Minuten einfuegen
100 #     df_new['co2_ppm_shift1'] = df_new.shift(1)['co2_ppm']
101 #     df_new['co2_ppm_shift2'] = df_new.shift(2)['co2_ppm']
102 #     df_new['co2_ppm_shift3'] = df_new.shift(3)['co2_ppm']
103 #     df_new['co2_ppm_shift4'] = df_new.shift(4)['co2_ppm']
104 #     df_new['co2_ppm_shift5'] = df_new.shift(5)['co2_ppm']
105 #     df_new['co2_ppm_shift6'] = df_new.shift(6)['co2_ppm']
106 #     df_new['co2_ppm_shift7'] = df_new.shift(7)['co2_ppm']
107 #     df_new['co2_ppm_shift8'] = df_new.shift(8)['co2_ppm']
108 #     df_new['co2_ppm_shift9'] = df_new.shift(9)['co2_ppm']
109 #     df_new['co2_ppm_shift10'] = df_new.shift(10)['co2_ppm']
110 #     df_new['co2_ppm_shift11'] = df_new.shift(11)['co2_ppm']
111 #     df_new['co2_ppm_shift12'] = df_new.shift(12)['co2_ppm']
112 #     df_new['co2_ppm_shift13'] = df_new.shift(13)['co2_ppm']
113 #     df_new['co2_ppm_shift14'] = df_new.shift(14)['co2_ppm']
114 #     df_new['co2_ppm_shift15'] = df_new.shift(15)['co2_ppm']
115
116 #     df_new['shift1_delta'] = df_new.shift(1)['co2_ppm'] - df_new.shift
        (2)['co2_ppm']
117 #     df_new['shift2_delta'] = df_new.shift(2)['co2_ppm'] - df_new.shift
        (3)['co2_ppm']
118 #     df_new['shift3_delta'] = df_new.shift(3)['co2_ppm'] - df_new.shift
        (4)['co2_ppm']
119 #     df_new['shift4_delta'] = df_new.shift(4)['co2_ppm'] - df_new.shift
        (5)['co2_ppm']
120 #     df_new['shift5_delta'] = df_new.shift(5)['co2_ppm'] - df_new.shift
        (6)['co2_ppm']
121 #     df_new['shift6_delta'] = df_new.shift(6)['co2_ppm'] - df_new.shift
        (7)['co2_ppm']
122
123     # Wochentag einfuegen
124     # verringert Genauigkeit, weil wahrscheinlich zu "verlaesslich"
125
126     df_new = insertWeekday(df_new)
127     #df_new = df_new.drop(df_new[df_new.dayOfWeek > 4].index)
128
129     # Leere Felder entfernen
130     df_new = df_new.dropna(axis=0, how='any', thresh=None, subset=None,
        inplace=False)
131
132     # Ausreisser mit Interquartile Range (IQR) und Tukey's Method
        loeschen
133     outlier_indices = detectOutliers(df_new)
134     df_new.drop(index=outlier_indices, inplace=True)
135
136     return df_new
137
138 def reshape_data_for_LSTM(X, y, timesteps_per_sample):
139     X = X.copy()
140     sample_count = int(X.count()[0]/timesteps_per_sample)
```

```
141
142     while X.count()[0] % sample_count != 0:
143         X = X.iloc[1: , :]
144         y = y.iloc[1:]
145
146     X_array = X.to_numpy()
147     y_array = y.to_numpy()
148
149     feature_count = len(X.columns)
150
151     print('sample_count = ' + str(sample_count))
152     print('timesteps_per_sample = ' + str(timesteps_per_sample))
153     print('feature_count = ' + str(feature_count))
154
155     X_reshaped = np.reshape(X_array,[sample_count, timesteps_per_sample
        , feature_count])
156     y_reshaped = np.reshape(y_array,[sample_count, timesteps_per_sample
        , 1])
157
158     return X_reshaped, y_reshaped
159
160 def normalize_min_max(dataframe, scaler=None):
161     if scaler is None:
162         scaler = MinMaxScaler(feature_range=(0, 1))
163         dataframe = pd.DataFrame(data=scaler.fit_transform(dataframe),
        columns=dataframe.columns)
164     else:
165         dataframe = pd.DataFrame(data=scaler.transform(dataframe),
        columns=dataframe.columns)
166     return dataframe, scaler
```

app/prepareData.py

```
1  import matplotlib.pyplot as mplt
2
3  def plotHistoryAccuracy(history):
4      mplt.plot(history.history['accuracy'])
5      mplt.plot(history.history['val_accuracy'])
6      mplt.title('model accuracy')
7      mplt.ylabel('accuracy')
8      mplt.xlabel('epoch')
9      mplt.legend(['train', 'val'], loc='lower right')
10     return mplt
11
12 def plotHistoryLoss(history):
13     mplt.plot(history.history['loss'])
14     mplt.plot(history.history['val_loss'])
15     mplt.title('model loss')
16     mplt.ylabel('loss')
17     mplt.xlabel('epoch')
18     mplt.legend(['train', 'val'], loc='upper right')
19     return mplt
```

app/validation.py

B. Rohdatenvisualisierungen

1. Graustufen
2. Verteilungen