



# Regenvorhersage in Australien

## Bericht

**Maschinelles Lernen mit Python**

vorgelegt von

**Kevin Settler**

E-Mail: Kevin.Settler@HS-Ansbach.de

Hochschule für angewandte Wissenschaften Ansbach

Master of Applied Research in Engineering Sciences

| | |
|---|---|
| Prüfer: | Johannes Dettelbacher, M.Sc. |
| | Dipl.-Ing. David Wagner |
| Abgabefrist: | 22.07.2022 |

# Inhaltsverzeichnis

# Abbildungsverzeichnis



# Tabellenverzeichnis

# 1 Ziel der Arbeit

Die Projektarbeit ist im Rahmen des Kurses „Einführung in maschinelles Lernen mit Python“ durchzuführen. Hiermit werden die Grundkenntnisse, die während der Vorlesungen übermittelt werden, verstärkt und zusätzlich vertieft.

Die Daten, die für diese Projektarbeit ausgehändigt wurden, liegen in einer csv-Datei vor und sollen dazu verwendet werden eine Wettervorhersage über Python zu generieren, der es möglich ist durch die angewendeten Methoden und Modelle die Entscheidung zu treffen, ob es am darauffolgenden Tag der betrachteten Daten regnen wird. Um mit dem Datensatz richtig umgehen zu können, wird sich im Vorfeld mit dessen Inhalt und Struktur vertraut gemacht.

# 2 Mit den Daten vertraut machen

Der zur Verfügung gestellte Datensatz enthält von ca. neun Jahren Wetterdaten, die in verschiedene Merkmale (= Features) unterteilt sind. Insgesamt liegt im originalen Zustand ein *DataFrame* der Größe 145.460 x 23 vor. Die Daten sind vereinzelt kategorisch wie der Ort (= *Location*), von welchem die Messdaten stammen oder Windrichtungen (= *WindDir*), aber auch nummerisch, wie Temperaturen (= *Temp*), relative Luftfeuchtigkeit (= *Humidity*), Luftdruck (= *Pressure*) und Bewölkung (= Cloud).

Über die pandas Funktion *describe()* wird ein statistischer Überblick der Daten gewährt. Wie in der Abbildung 2-1 zu sehen ist, werden verschiedene Merkmale, wie der insgesamten Anzahl (= *count*), der Anzahl verschiedener Werte (= *unique*), dem Mittelwert (= *mean*) für jede einzelne Spalte (= *column*) des *DataFrame*, angetragen [1].

```
original_data.describe(include = "all") #count ist ungleich --> Anzahl der Daten sollte überall gleich sein
```

| | Date | Location | MinTemp | MaxTemp | Rainfall | Evaporation | Sunshine | WindGustDir | WindGustSpeed | WindDir9am | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 145460 | 145460 | 143975.000000 | 144199.000000 | 142199.000000 | 82670.000000 | 75625.000000 | 135134 | 135197.000000 | 134894 | ... |
| unique | 3436 | 49 | NaN | NaN | NaN | NaN | NaN | 16 | NaN | 16 | ... |
| top | 2013-11-12 | Canberra | NaN | NaN | NaN | NaN | NaN | W | NaN | N | ... |
| freq | 49 | 3436 | NaN | NaN | NaN | NaN | NaN | 9915 | NaN | 11758 | ... |
| mean | NaN | NaN | 12.194034 | 23.221348 | 2.360918 | 5.468232 | 7.611178 | NaN | 40.035230 | NaN | ... |
| std | NaN | NaN | 6.398495 | 7.119049 | 8.478060 | 4.193704 | 3.785483 | NaN | 13.607062 | NaN | ... |
| min | NaN | NaN | -8.500000 | -4.800000 | 0.000000 | 0.000000 | 0.000000 | NaN | 6.000000 | NaN | ... |
| 25% | NaN | NaN | 7.600000 | 17.900000 | 0.000000 | 2.600000 | 4.800000 | NaN | 31.000000 | NaN | ... |
| 50% | NaN | NaN | 12.000000 | 22.600000 | 0.000000 | 4.800000 | 8.400000 | NaN | 39.000000 | NaN | ... |
| 75% | NaN | NaN | 16.900000 | 28.200000 | 0.800000 | 7.400000 | 10.600000 | NaN | 48.000000 | NaN | ... |
| max | NaN | NaN | 33.900000 | 48.100000 | 371.000000 | 145.000000 | 14.500000 | NaN | 135.000000 | NaN | ... |

11 rows × 23 columns

Abbildung 2-1: statistischer Überblick der Originaldaten

Der *count* gibt die insgesamte Anzahl an verwertbare *Features* wieder. Es ist schlecht, dass die *columns* nicht einheitlich viele Werte haben, die für das Trainieren und Testen der Modelle verwendet werden kann.

Nachdem ein Überblick über die vorliegenden Daten gemacht wurde, kann damit weiter verfahren werden diese so zu verändern und anzupassen, sodass sie für spätere Modellbildungen und Funktionen kompatibel sind und zu keinen Fehlermeldungen führen.

# 3 Zur Modellbildung aufbereiten

Auffallend ist, dass viele Werte im *DataFrame* mit *NaN* ausgefüllt sind. Dies steht ausgeschrieben für „*Not a Number*“. *NaN* steht in Python an jenen Stellen, an denen das *Feature* undefiniert ist bzw. gar kein Wert hinterlegt ist. In der Modellbildung ist ein solcher Ausdruck hinderlich, da dadurch die gesamte Datenreihe ihre Aussagekraft verliert.

Die Daten sind in ihrer gängigen Einheit (bspw.: Temperatur in Grad Celsius) angegeben. Für den Menschen ist dies zwar dienlich, der Computer kann damit aber weniger gut umgehen. Deshalb werden die Daten skaliert. Konkret werden sie standardisiert und anschließend normalisiert.

Die kategorischen Daten sind für das spätere Vorgehen ebenfalls in nummerische Daten umzuwandeln. Umgesetzt wird dies für die *Location*, *WindGustDir*, *WindDir9am*, und *WindDir3pm*.

In den folgenden Abschnitten des Dokuments wird erklärt, wie mit den genannten Auffälligkeiten umgegangen wird, um das *DataFrame* für das maschinelle Lernen aufzubereiten.

## 3.1 Alle unbekannten Werte zu NaN umwandeln

Um eine Gesamtheit jener *Features*, die nicht verarbeitet werden können, zu erhalten, werden alle kritischen Werte zu *NaN* umgewandelt. Als kritische Werte, die mit *NaN* ersetzt werden müssen, werden das Fragezeichen, eine Leertaste und ein Bindestrich erachtet.

## 3.2 Umgang mit NaN

Zur besseren Überwachung wird die Funktion *count_NaN* erstellt. Diese zählt die absolute Anzahl an *NaN* für jede *column* und gibt zustzlich den Anteil, bezogen auf die Gesamtheit der Werte in dieser column in einer Tabelle zurück. In den Originaldaten kommen insgesamt 343.248 *NaN* vor, was einen Anteil von 10,26 % ausmacht. Als anfängliche Lösung werden die *NaN* in den nummerischen Daten mit dem Median der jeweiligen *column* ersetzt. Es wurde sich für den Median entschieden, um sich der Mehrheit der authentischen Originaldaten anzupassen. Würde man die ca. 10 % der Daten mit bspw. einer Null ersetzen,

käme es zu einer Verzehrung des Mittelwertes, durch eine zur Null hin ungleichmäßig hohen Verteilung der Werte.

Die *NaN* innerhalb der kategorischen Daten können nicht mit einem Median ersetzt werden, da dieser nur für nummerische Werte generiert werden kann. Um der Logik, dass man die Verteilung möglichst wenig verzehren möchte, beizubehalten, werden sie mit jenem Wert ersetzt, der am häufigsten innerhalb der jeweiligen *column* vorkommt.

Die beiden *Features RainToday* und *RainTomorrow* stellen einen Sonderfall dar. Wohingegen *RainToday* ein vermeintlich gewichtiges *Feature* zum Trainieren des Modelles ist, ist *RainTomorrow* das Ziel (= *Target*) der zu erstellenden Modelle. Darüber hinaus muss *RainToday* als besonderes *Feature* behandelt werden, weil die Option, es als zusätzliches *Target* zu verwenden, offengehalten werden soll. Aufgrund dieser Bedeutsamkeit ist davon abzuraten, ungültige Werte mit irgendetwas zu ersetzen, da dies zu Verfälschungen führen kann. Hinsichtlich des Anteils innerhalb der beiden *columns* von ca. 2,25 % wird sich dazu entschieden, bei einem *NaN* in einem der beiden columns die gesamte Reihe (= *row*) aus dem Datensatz zu entfernen.

Durch das Löschen von bestimmten *rows* verläuft der Index, der die Anzahl der Datenreihen zählt, nicht mehr kontinuierlich. Um das Erzeugen neuer *NaN* zu vermeiden, wird der Index zurückgesetzt, sodass die *rows* erneut von Null an aufsteigend nummeriert werden.

## 3.3 Daten skalieren

Die Standardisierung weist den Daten einen nummerischen Wert zwischen -1 und 1 zu. Dies ist wichtig, weil unterschiedliche Größen mit verschiedenen Einheiten verglichen werden. So wird der Luftdruck bspw. in der Größenordnung von $1 \cdot 10^3$ hPa und die Temperaturen bis maximal 50 °C angegeben. Beim Trainieren im Modell würden die Werte des Luftdrucks aufgrund des höheren Wertes, eine größere Gewichtung zuteilwerden, was zu verhindern ist.

Der Grund für das Normalisieren ist ein ähnlicher. Es sind die Werte numerischer columns im Datensatz auf eine gemeinsame Skala zu transformieren. Hierbei dürfen jedoch keine Unterschiede in den Wertebereichen verzehrt werden, um die Authentizität der Daten nicht zu verfälschen [2].

Das Durchführen dieser beider Skalierungen wird in Funktionen zusammengefasst.

## 3.4 One-Hot Encoding

Vor dem One-Hot Encoding wurden zwei Lexika (= *dictionaries*) erzeugt. Eines für die kategorischen und das andere für die nummerischen Daten. Diese werden mit den Werten der jeweiligen *Features* gefüllt (bspw. bei *Location* die Namen der verschiedenen *Locations*).

Darüber hinaus wird eine zusätzliche *column*, die *RainySeason*, in das *DataFrame* implementiert. Diese vergibt, aufgrund der Einträgen für das Datum (= *Date*) nummerische Werte, welche Jahreszeit in Australien zu diesem Zeitpunkt vorgeherrscht hat. Auf dieser Grundlage sollte eine weitere Klassifikation möglich sein. Hiermit kann ersichtlich gemacht werden, zu welcher Jahreszeit es in Australien mehr oder weniger regnet.

Der *OneHotEncoder* erzeugt für jedes neue Feature innerhalb einer *column* eine eigene *column*. Dort werden dann Nuller oder respektive Einser eingetragen, je nachdem, ob dieser Feature in dieser *row* wahr oder falsch ist [3]. Konkret an einem Beispiel bedeutet das, dass es der Encoder für eine neue *Location* eine neue *column* erzeugt. Dieser werden dann in den *rows*, in denen sie bei der *column Location* stand, eine Eins zugeordnet. In allen anderen stehen dann Nullen.

Der *OneHotEncoder* wird separat auf den Bereich der kategorischen Daten und die *RainySeason* angewendet.

Der *OneHotEncoder* vergibt den neuen *columns* nicht automatisch neue Namen. Aufgrund dessen wurden zuvor die *dictionaries* erstellt. Diese werden über eine *for*-Schleife den erstellten *columns* zugewiesen, sodass sie eindeutige Namen tragen.

Am Ende des *One-Hot Encoding* ergibt sich ein 140.787 x 121 *DataFrame*.

Anschließend werden jene Daten, die nicht dem Datentyp *float* entsprechen, in diesen umgewandelt, sodass das gesamte *DataFrame* aus einem einheitlichen Datentyp besteht.

# 4 Daten analysieren

Für die Datenanalyse wird ein *Pairplot* auf Grundlage der Originaldaten erstellt. Dieser ist innerhalb des Codes auskommentiert, da die Generierung zeit- und rechenaufwendig ist. Die fertig erzeugte Datei liegt als *portable network graphic* im Anhang dem Bericht bei.

Um zu sehen, wie die einzelnen *Features* miteinander korrelieren, wird eine *Heatmap* aus den mittlerweile kodierten Daten erstellt. Eine *Heatmap* ist eine Funktion auf Achsenebene und stellt, optional mit einer zusätzlichen Farbskala, die Korrelation zwischen den *Features* dar [2]. In der Abbildung 4-1 sind *Features*, die miteinander korrelieren, diejenigen, bei denen ein Wert nahe der Eins angetragen ist. Umso höher der Wert, desto höher die Korrelation. Zu erkennen ist das besonders an den Stellen, an denen auf beiden Achsen das gleiche *Feature* angetragen ist. Die Korrelation entspricht dabei einer Eins. Dies bedeutet bspw. für das *Feature Temp3pm*, dass die Temperaturen gleich sind, was aufgrund desselben Features selbstredend ist. Vergleicht man hingegen die Korrelation mit dem *Feature Temp9am*, erkennt man einen Wert von 0,85. Hieraus lässt sich interpretieren, dass die Temperaturen zwischen 09:00 Uhr vormittags und 15:00 Uhr nachmittags oft sehr ähnlich sind.

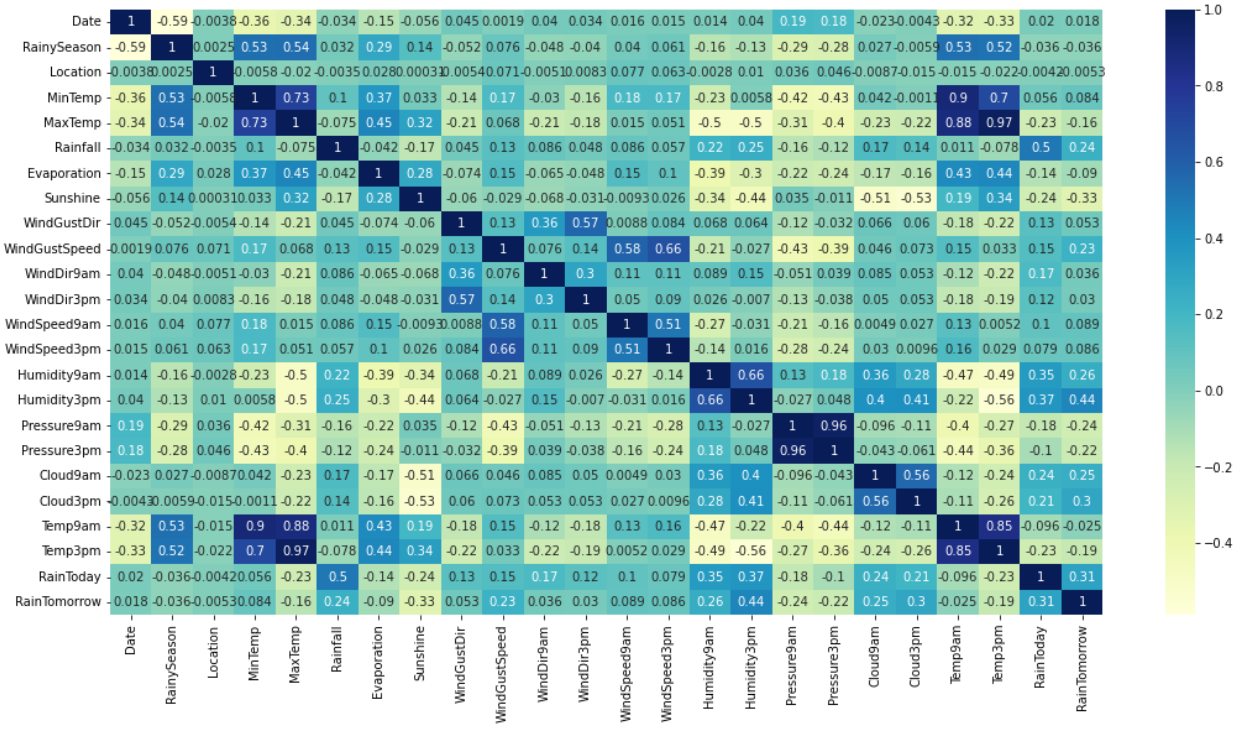


Abbildung 4-1: *Heatmap* zur Korrelation der kodierten Daten

Miteinander korrelierende Daten können zusammengefasst werden, da sie ähnliche Auswirkungen auf das *target* haben. Zudem kann somit die Anzahl der gesamten *Features* reduziert werden, was die Rechenzeit beschleunigt. In diesem Fall werden die Daten wie folgt zusammengefasst:

- $MaxTemp = MaxTemp \cdot Temp3pm$
- $MinTemp = MinTemp \cdot Temp9am$
- $Pressure = Pressure9am \cdot Pressure3am$

Anschließend sind die verbleibenden *Features*, die nicht überschrieben wurden, aus dem *DataFrame* zu entfernen.

Die *heatmap* wird erneut ausgegeben, um die Wirkung des Vorgehens zu kontrollieren. In der Abbildung 4-2 ist, abgesehen von jedem *Feature* zu sich selbst, keine Korrelation von über 80 % mehr zu finden.

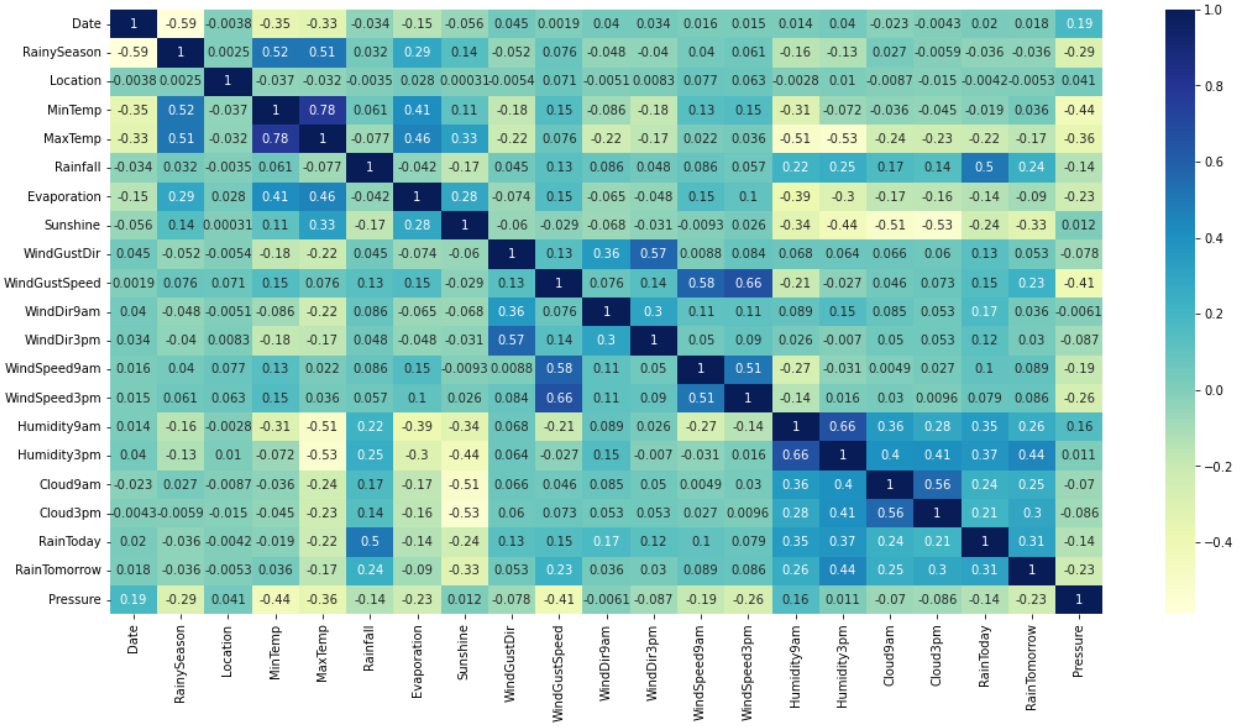


Abbildung 4-2: *Heatmap* zur Korrelation der kodierten Daten nach Zusammenfügen der korrelierenden Daten

Ebenfalls mit den kodierten Daten werden neben der *Heatmap* zusätzlich *histograms* erzeugt. Hierbei ist zu erwähnen, dass dieser Schritt mit dem Datensatz von vor dem Normalisieren und Standardisieren der Daten durchgeführt wird. Diese ermöglichen es die Verteilung der Daten einzusehen [3]. In der Abbildung 4-3 sind die *histograms* der *Features* gesammelt. Es ist bei den *Features MinTemp*, *MaxTemp*, *WindGustSpeed*, *WindSpeed9am*, *WindSpeed3pm*, *Humidity9am*, *Humidity3pm*, *Pressure9am*, *Pressure3pm*, *Temp9am* und *Temp3pm* zumindest grob eine Normalverteilung zu erkennen.

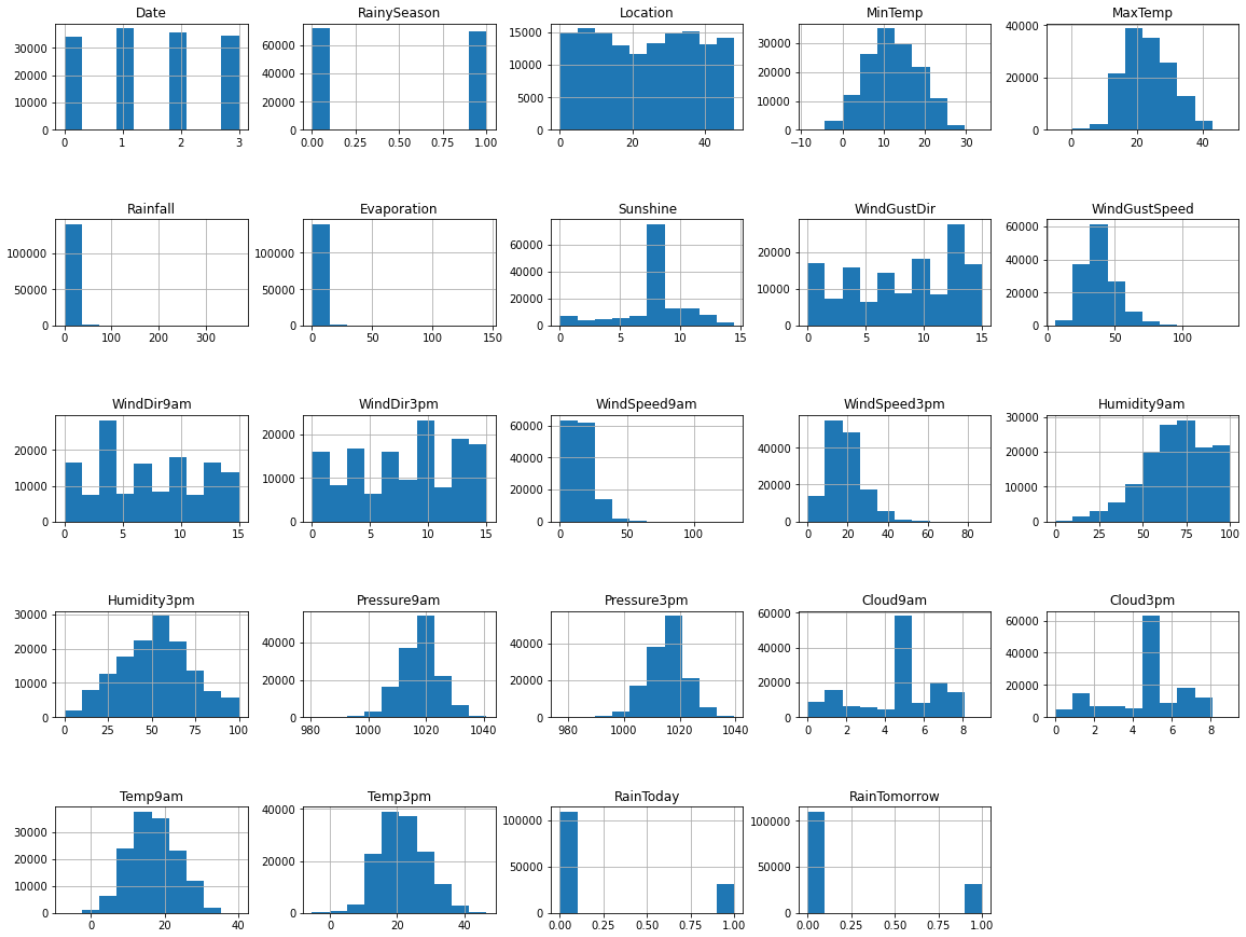


Abbildung 4-3: *Histograms* zur Korrelation der kodierten Daten

# 5 Anwenden unterschiedlicher Modelle

Für das Anwenden von Modellen müssen die Daten in Test- und Trainingsdaten aufgeteilt werden. Es wird sich dazu entschieden 80 % der Daten als Trainingsdaten zu verwenden und den Rest als Testdaten. Grund hierfür ist, dass die Modelle mit den Trainingsdaten lernen sollen und anschließend fähig sein sollten Vorhersagen zu treffen. Diese Vorhersagen werden auf Grundlage der verbleibenden Testfeatures gemacht und anhand der dazugehörigen Testtargets validiert.

*Features* werden dem x zugewiesen, während y das *target* repräsentiert.

Für das Optimieren der verschiedenen Modelle werden die Trainingsdaten erneut aufgeteilt, um zu verhindern, dass die Testdaten in irgendeiner Form verwendet werden. Diese sollen unverbraucht bleiben, sodass sie für das Modell gänzlich unbekannt sind, wenn man es zuletzt mit den gesamten Daten durchführt.

## 5.1 Decision Tree

Zuerst wird der *DecisionTreeClassifier()* verwendet. Die hierfür eingestellten *hyperparameter* sind die Folgenden [6]:

- *max_depth* = 8 (maximale Anzahl an Ebenen)
- *min_samples_leaf* = 5 (minimale Anzahl, die für ein Blatt benötigt wird)
- *min_samples_split* = 3 (minimale Anzahl, die für eine Verzweigung benötigt wird)

Nach der Erstellung des Modells werden die *x/y_train_optimize* Daten verwendet. Nachdem implementiert wird, für welche Daten Vorhersagen getroffen werden sollen, wird die Genauigkeit (= *accuracy*) für den Trainings- und Testdurchlauf berechnet.

## 5.2 Naive Bayes

Anschließend wird ein *Naive Bayes* Modell mit dem *GaussianNB()* aufgebaut. Die *hyperparameter* dieses *classifiers* werden nicht abgeändert.

Hierbei ist darauf zu achten, dass die Version 2.9.1 von *TensorFlow* auf dem ausführenden Gerät installiert ist.[1]

Nach der Erstellung des Modells ist das weitere Vorgehen analog zu dem im Abschnitt 5.1.

[1] Mit der Version 2.9.1 von TensorFlow läuft der Code in VS Code komplett und fehlerfrei durch.
In Google Colab lief er nicht durch und hat sich an dieser Stelle gestört (da war eine ältere Version installiert)

# 6 Ergebnisse der Modelle miteinander vergleichen

Die zuvor berechnete *accuracy* wird ausgegeben und für beide Modelle gegenübergestellt. In der Tabelle 6-1 ist zu erkennen, dass der *Decision Tree* sowohl beim Trainieren des Modells als auch beim Test besser abschneidet als das *Naive Bayes* Modell.

Tabelle 6-1: *train-/test_accuracy* von *Decision Tree* und *Naive Bayes*

| Modell | *train_accuracy* | *test_accuracy* |
|---|---|---|
| *Decision Tree* | 0,8531458442005261 | 0,8412945041285625 |
| *Naive Bayes* | 0,6555275629002364 | 0,7842049187605433 |

Darüber hinaus wird die Fehlerrate für beide Modelle in Form des *mean absolute error* (= *MAE*), mean square error (= *MSE*) und root mean square error (= *RMSE*) bestimmt.

# 7 Hyperparameter optimieren

Die *hyperparameter* werden für die beiden Modelle auf unterschiedliche Weise optimiert. Während der *Decision Tree* mittels der Methode *GridSearchCV* untersucht wird, wird beim Naive Bayes Modell *TensorFlow* verwendet, um die *hyperparameter* zu variieren und eine möglichst hohe Genauigkeit für die Modelle zu erreichen.

## 7.1 Decision Tree: GridSearchCV

Die drei ausgewählten *hyperparameter* werden unter den folgenden Werten variiert:

- *max_depth* = 2, 4, 5, 6, 8
- *min_samples_leaf* = 2, 4, 6, 8, 10
- *min_samples_split* = 2, 4, 6, 8, 10

Die drei besten *hyperparameter*, die hierdurch ermittelt wurden, werden in eine Variable übergegeben. Anschließend wird das Modell des *Decision Tree* erneut durchgeführt, wobei die vorher erstellte Variable in die Funktion eingesetzt wird, sodass diese Einstellungen für das optimierte Modell verwendet werden.

### 7.2 Naive Bayes: TensorFlow

Für das Optimieren über *TensorFlow* werden alle Parameter, die zuvor ausgewählt wurden und somit optimiert werden können, miteinander kombiniert, um eine möglichst hohe Genauigkeit auf die Vorhersagen des Modells zu erreichen. Mit der getroffenen Auswahl können insgesamt 1 086 Parameter optimiert werden. Das Modell wird anschließend mit einer batch_size von 20, damit der Zeitaufwand etwas reduziert werden kann, über 200 Epochen optimiert.

## 8 Durchführen des gesamten Modells

Aufgrund eines Problems des *overfittings* beim Naive Bayes für kleinere Datenmengen (bspw. für *test_size* = 0.99) und der besseren Aufbereitung des *Decision Tree* durch die Funktion der *GridSearchCV* werden das Modell des *Decision Tree* auf die Gesamtheit der vorliegenden Daten angewendet. Folglich werden *x_train*, *y_train*, *x_test* und *y_test* und nicht mehr die zweifach aufgeteilten Daten.

Als *hyperparameter* werden die optimierten Werte eingesetzt.

Die entstehenden Bewertungskriterien für das Modell sind in der Tabelle 8-1 zusammengefasst.

Tabelle 8-1: Bewertungsmerkmale des Decision Tree auf die Gesamtheit der Daten

| Bewertungsmerkmal | Wert |
|---|---|
| *train_accuracy* | 0,8510507950882987 |
| *test_accuracy* | 0,8415370409830244 |
| *MAE* | 0,15846295901697563 |
| *MSE* | 0,15846295901697563 |
| *RMSE* | 0,3980740622258321 |

## 9 Diskussion

Zu hinterfragen sind noch folgende Punkte:

- Zusammenfassen der miteinander korrelierenden Features (Multiplizieren scheint bedingt sinnvoll)
- Wieso ist die *train*- und *test_accuracy* beim optimierten *GridSearchCV* gleich denen von Beginn an?
    - Zudem wurde festgestellt, dass bei jedem Durchgang die Bewertungsmerkmale verschieden sind)

# Literaturverzeichnis

[1] pandas development team, „pandas documentation,“ [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.describe.html. [Zugriff am 22 07 2022].

[2] „ichi.pro,“ [Online]. Available: https://ichi.pro/de/wie-wann-und-warum-sollten-sie-ihre-daten-normalisieren-standardisieren-neu-skalieren-69304631376127. [Zugriff am 22 07 2022].

[3] skikit-learn developers, „skikit-learn documentation,“ [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html. [Zugriff am 22 07 2022].

[4] M. Waskom, „seaborn documentation,“ [Online]. Available: http://seaborn.pydata.org/generated/seaborn.heatmap.html. [Zugriff am 22 07 2022].

[5] J. Hunter, D. Dale, E. Firing und M. Droettboom, „matplotlib documentation,“ [Online]. Available: https://matplotlib.org/stable/gallery/statistics/hist.html. [Zugriff am 22 07 2022].

[6] skikit-learn developers, „scikit-learn documentation,“ [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html. [Zugriff am 22 07 2022].

# Anhang:

- pairplots_corr_original_data.png