Hospitalization Model

AKANW: Projektpraktikum aus Techn.Mathematik

Aleksandar Dadic

10. Oktober 2022

1 Aufgabenstellung

Es ist klar, dass die Zahl der COVID-19 Hospitalisierungen stark von den Infektionszahlen abhängt. Umgekehrt hat aber die Krankenhausauslastung nur einen vergleichsweise geringen Einfluss auf die Ausbreitung der Krankheit. Diese Tatsache motiviert die Strategie, ein epidemiologisches Modell zu verwenden, um zunächst die Fallzahlen zu prognostizieren, und dann ein anderes Modell zu verwenden, um die Hospitalisierungen auf der Basis der simulierten Fallzahlen vorherzusagen. Der Vorteil dabei ist, dass so beide Modellierungsprozesse einschließlich aller Probleme mit der Verifikation, Parametrisierung und Validierung voneinander getrennt behandelt werden können.

Das Ziel dieses Projektes ist es so ein Hospitalisierungsmodell zu entwerfen. Das Modell soll also in der Lage sein, die COVID-19-bedingte Normal- und Intensivbettenbelegung mithilfe von gegebenen und/oder prognostizierten Neuinfektionszahlen vorherzusagen. Dazu werden zwei Ansätze verglichen: ein kausales Modellierungskonzept und ein rein datenbasierter Ansatz unter Verwendung statistischer Konzepte. Beide Modelle wurden in Python implementiert.

2 Daten für Fallzahlen und Hospitalisierungen

Beide Modellierungsansätze benötigen Fallzahl- und Hospitalisierungsdaten. Als Quelle dafür dienen die täglich aktualisierten Daten der AGES. Anstelle der regulären Fallzahldaten verwenden wir jenen Datensatz[1], der die Anzahl der COVID-Infektionen neben den Bundesländern auch nach Altersgruppe aufschlüsselt. Dies ermöglicht es, uns bei der Prognose der Krankenhausauslastung auf ältere Altersgruppen zu beschränken, die laut [BRZ] hauptverantwortlich für die Belegung von Spitalsbetten sind. Die Fallzahldaten sind ab 26.02.2020 dokumentiert. Eine größere Einschränkung sind die Hospitalisierungsdaten[2], welche erst ab 24.01.2021 auf der AGES Seite verfügbar sind. Um den Zeitraum

[1] `https://covid19-dashboard.ages.at/data/CovidFaelle_Altersgruppe.csv`
[2] `https://covid19-dashboard.ages.at/data/Hospitalisierung.csv`

zu vergrößern, wurden mir von meinem Betreuer Martin Bicher nicht-öffentliche Hospitalisierungsdaten bereitgestellt, welche bis zum 10.11.2020 zurückgehen. Für die Fallzahl- und Hospitalisierungsdaten wurde das Python Modul `data_loader` mit den Klassen `OccupancyData` und `CaseData` geschrieben, welche die spätere Verwendung der Daten in den Modellen vereinfacht.

3 Kausales Modell

Das kausale Modell beruht auf dem Ansatz, dass jede am Tag t infizierte Person mit der Wahrscheinlichkeit p am Tag $t + T_1$ hospitalisiert wird und am Tag $t + T_1 + T_2$ aus dem Krankenhaus entlassen wird. Die Zeitspannen T_1 und T_2 sind dabei Zufallsvariablen, deren Verteilungen zunächst festgelegt werden muss. Das Problem dabei ist, dass es praktisch keine Daten oder Quellen gibt, die Aufschluss darüber geben könnten, wie T_1 und T_2 verteilt sind. Da T_2 die Verweildauer ist, welche für die Beurteilung der Krankenhausauslastung ein interessanter Wert ist, gibt es dazu ein paar Infos. Allerdings findet man gewöhnlich nur Daten zur mittleren Verweildauer (z.B. in [BRZ]), welche natürlich zur Bestimmung der Verteilung nicht hilfreich ist. Ich konnte nur ein Paper finden ([RNJ+20]), welches sich mit der Verteilung der Verweildauer beschäftigt. Folgende Grafik stammt aus jenem Paper:

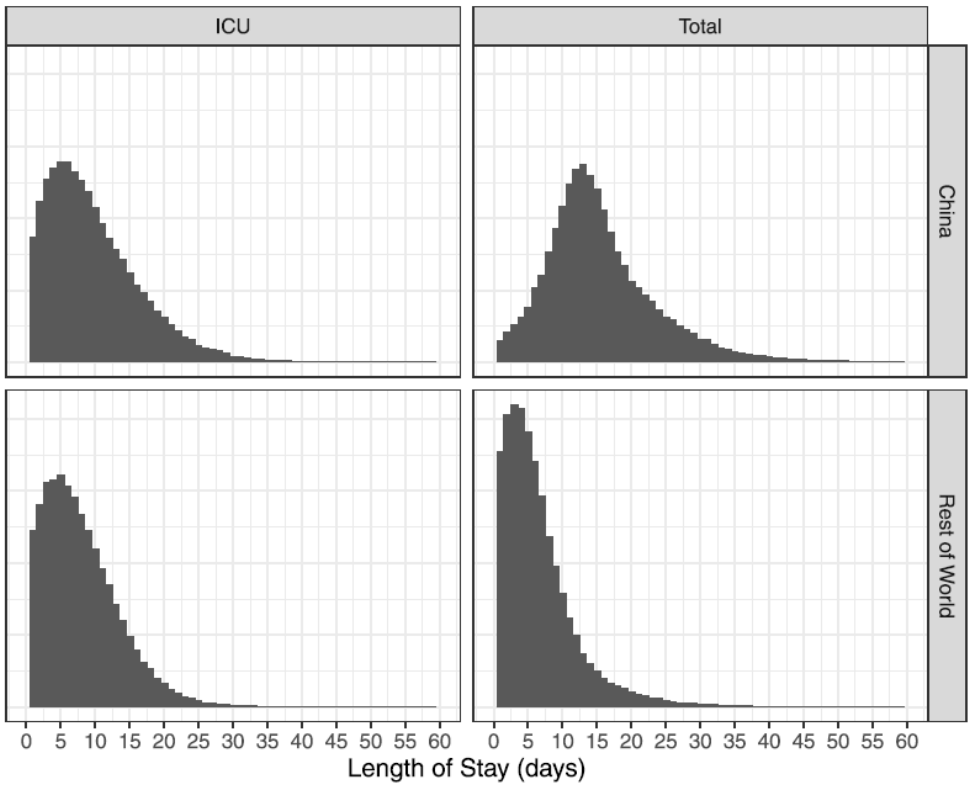

Abbildung 1: Verteilungen der Verweildauer (aus [RNJ+20])

Die Form erinnert an eine auf die positive Achse beschränkte Normalverteilung (*truncated normal distribution*[3]), weshalb wir diese Verteilung für T_2 wählen. Wie die „normale“ Normalverteilung ist die *truncated normal distribution* durch den Mittelwert μ und die Varianz σ^2 bestimmt. Im Gegensatz dazu gibt es für die Verteilung von T_1 überhaupt keine Anhaltspunkte, was vermutlich auch daran liegt, dass Daten zur Zeitspanne zwischen Infektion und Krankenhausaufenthalt eher schwierig zu erheben sind. Da die Poission-Verteilung nur einen einzigen Parameter λ hat, wurde daher relativ willkürlich $T_1 \sim \text{Pois}(\lambda)$ festgelegt, in der Hoffnung, dass eventuelle Diskrepanzen in der Kalibrierung ausgeglichen werden. Das kausale Modell hat somit die 4 Parameter $(p, \mu, \sigma^2, \lambda)$. Die Python-Implementierung dieses Modells besteht aus der Klasse `CausalModel` im Modul `causal_model`, deren Verwendung im Quellcode beschrieben wird. Die nachfolgenden Unterabschnitte behandeln die Implementierung dieser Klasse.

3.1 Simulation

Die Implementierung des oben beschriebenen Schemas ist relativ einfach, wie folgender Pseudo-Code zeigt:

Algorithm 1: Simulation des kausalen Modells

Input: Parameter $(p, \mu, \sigma^2, \lambda)$ und Vektor $\mathbf{v}$ von Fallzahlen der Länge n
Output: Vektor $\mathbf{w}$ der Länge n mit simulierten Belagszahlen

```
foreach t ∈ {1, ..., n} do
    m := v[t]
    foreach i ∈ {1, ..., m} do
        t1 := Realisierung von T1
        t2 := Realisierung von T2
        a := t + t1
        b := t + t1 + t2
        w[a ... b] += 1
    end
end
```

Eine Herausforderung war allerdings, den Code möglichst effizient zu implementieren. Dies wurde durch zwei Maßnahmen erreicht:

- Verwendung von `numpy`-Vektoren für $\mathbf{v}$ und $\mathbf{w}$
- Einmalige Realisierung großer Zufallsvektoren für T_1 und T_2 vor den Schleifen

Ein weiteres Problem stellt die Initialisierung des Vektors $\mathbf{w}$ dar. Es ist zwar naheliegend $\mathbf{w}$ als Nullvektor anzulegen; weil aber die Fallzahlen $\mathbf{v}$ zeitversetzt auf $\mathbf{w}$ wirken, führt dies dazu, dass die ersten Einträge von $\mathbf{w}$ sinnlos sind. Da wir aber annehmen können, dass die Fallzahlen nur in einem überschaubaren Zeitraum einen Einfluss auf die Hopsitalisierungen haben, sollten die Einträge von $\mathbf{w}$ ab einem Index sinnvoll sein.

[3] Truncated normal distribution

Das motiviert die Strategie, die Simulation entsprechend vor dem gewünschten Simulationsintervall $[t_0, t_1]$ zu starten, und $\mathbf{w}$ dann im Anschluss auf $[t_0, t_1]$ einzuschränken. Wie viel früher die Simulation gestartet werden soll, wird in der `CausalModel` Klasse mit dem Parameter `buffer` eingestellt. In diesem Projekt ist dieser Wert stets auf 60 Tage festgelegt. Da die durchschnittliche Verweildauer laut [BRZ] bei lediglich 12,5 (ICU) bzw. 11,3 Tagen (Normalstation) liegt, sollte dieser „Puffer“ mehr als ausreichend sein.

3.2 Monte-Carlo Simulation

Die Implementierung einer Monte-Carlo Simulation ist bekanntermaßen sehr einfach: es müssen nur die Ergebnisse mehrerer Simulationen gemittelt werden. Da dabei alle Simulationen unabhängig voneinander stattfinden, lässt sich der Vorgang relativ einfach parallelisieren und somit massiv beschleunigen. Die Schwierigkeit dabei ist aber, dass damit im Normalfall trotz Festlegung des *seeds* die Reproduzierbarkeit verloren geht. Die Lösung für dieses Problem wird in der `numpy` Dokumentation[4] behandelt: Die Idee besteht darin, die einzelnen Simulationen zu nummerieren (= *worker id*) und einen sogenannten *root seed* festzulegen. Zu Beginn jeder Simulation wird dann der *root seed* mit der *worker id* der Simulation zu einem neuen *seed* kombiniert und damit ein neuer *random number generator* speziell für die Simulation erstellt. Somit hat jede Simulation einen eigenen einzigartigen *stream* von Zufallszahlen. Da die Nummerierung der Simulationen deterministisch ist, ist das Resultat aller Simulationen reproduzierbar, und somit auch die Monte-Carlo Simulation.

3.3 Kalibrierung

Um das Modell kalibrieren zu können, benötigen wir zunächst eine Fehlerfunktion. Wir haben dafür den *mean absolute percentage error* (MAPE) verwendet, welcher für die Simulation $\mathbf{x} = (x_1, \dots, x_n)$ und die Referenz $\mathbf{y} = (y_1, \dots, y_n)$ durch

$$MAPE := \frac{100}{n} \sum_{i=1}^{n} \left| \frac{x_i - y_i}{y_i} \right|$$

definiert ist. Das Ziel in der Kalibrierung ist also die Parameter $(p, \mu, \sigma^2, \lambda)$ so zu bestimmen, dass der MAPE möglichst klein ist. Um dieses Minimierungsproblem zu lösen, verwenden wir `scipy.optimize`. Zunächst wurde die Kalibrierung mittels `minimize` probiert. Diese Funktion hat einen Parameter `method`, der den Optimierungsalgorithmus bestimmt. Dabei hat sich herausgestellt, dass „Nelder-Mead“ im Vergleich zu den anderen Methoden die besten Ergebnisse liefert. Diese Resultate waren jedoch trotzdem nicht zufriedenstellend, was vermutlich daran liegt, dass `minimize` für lokale Optimierung gedacht ist und unser Minimierungsproblem scheinbar viele lokale Minima hat. Einen Anhaltspunkt für diese Behauptung liefern die Algorithmen für globale Optimierung, da diese viel bessere Ergebnisse hatten als Nelder-Mead. Unter diesen Algorithmen hat sich `differential evolution` als der beste Optimierungsalgorithmus herausgestellt.

[4] `https://numpy.org/devdocs/reference/random/parallel.html#sequence-of-integer-seeds`

Differential evolution[5] ist ein evolutionärer Algorithmus, der besonders gut für globale Optimierungsprobleme geeignet ist. Der *differential evolution* Algorithmus hat viele Parameter; ein wichtiger ist `strategy`, der die Art, wie neue Kandidaten erzeugt werden, bestimmt. Nach einigen Tests hat sich die Strategie „`randtobest1bin`“ bewährt. Die Belegung der anderen Parameter kann dem Quellcode entnommen werden. Ein Beispiel für ein kalibriertes Modell ist in Abbildung 14 dargestellt.

3.4 Laufzeit

Es ist klar, dass die Laufzeiten der Simulation und der Kalibrierung sehr stark von den Parametern, dem gewählten Zeitraum und der eigenen Hardware abhängen. Trotzdem wollen wir hier ein paar Laufzeiten zur Orientierung angeben. Dafür haben wir folgendes Szenario verwendet:

- Bundesland: Österreich
- Betten: Intensivstation
- Altersgruppen: 45+
- Buffer: 60 Tage
- Zeitraum: 01.03.2021 bis 30.03.2021 (30 Tage)

Die Kalibrierung dieses Modells braucht auf meinem Rechner ca. 5 Minuten, was aber vergleichsweise eher lang ist: die Laufzeit variiert sehr stark, je nach dem wann der *differential evolution* Algorithmus terminiert. In diesem Szenario ist der Algorithmus nicht von alleine terminiert, sondern wurde abgebrochen, weil die maximale Anzahl an Iterationen erreicht wurde (die wir auf 150 gesetzt haben). Terminiert der Algorithmus früher, kann eine Kalibrierung auch nur eine Minute dauern. Eine Simulation des kalibrierten Modells braucht ca. 6 ms und eine sequentielle Monte-Carlo Simulation mit $n = 15$ entsprechend ca. 90 ms. Eine parallele Monte-Carlo Simulation braucht im Gegensatz dazu mehr als 0,8 Sekunden, also ca. 9 Mal so lang. Der Grund dafür ist, dass *parallel computing* immer einen gewissen *overhead* hat, der in diesem Fall die Vorteile zunichte macht. Bei längeren Zeiträumen, mehr täglichen Neuinfektionen oder einem größeren n zahlt sich allerdings die parallele Monte-Carlo Simulation durchaus aus.

4 Datenbasiertes Modell

Um eine Prognose für eine Zeitreihe zu erstellen, gibt es grundsätzlich drei Ansätze[6]:

- Statistische Modelle (z.B. ARIMA, exponentielle Glättung, etc.)
- Machine Learning (z.B. Regression, XGBoost, Random Forest, etc.)

[5] `https://en.wikipedia.org/wiki/Differential_evolution`
[6] `https://www.datacamp.com/tutorial/tutorial-time-series-forecasting`

- Deep Learning (neuronale Netze wie RNN oder LTSM)

Wir werden uns in diesem Projekt auf die ersten beiden Ansätze beschränken. Um möglichst viele Modelle auszuprobieren, wurde das Python package `PyCaret` verwendet. `PyCaret` ist eine open-source, low-code Bibliothek für Machine Learning, die automatisch verschiedenste Modelle vergleichen kann, und somit dabei hilft, das beste Modell zu finden. Zuerst wurden die Modelle aus `pycaret.regression` ausprobiert, wofür zunächst *features* festgelegt werden mussten. Naheliegend ist natürlich dafür die täglichen Neuinfektionen zu nehmen, allerdings macht es unter Umständen Sinn diese Daten geeignet zu transformieren. In unserem Fall hat sich herausgestellt, dass wir mit mittels *rolling mean* geglätteten Fallzahlen bessere Ergebnisse erhalten. Für die `PyCaret` Tests haben wir die österreichweiten Fallzahl- und ICU-Belegungsdaten von 30.03.2021 bis 30.03.2022 verwendet. In Abbildung 2 sehen wir die nach MAPE sortierte Leistung der Modelle, die von `PyCaret` als Durchschnitt von 10 *folds* berechnet wurde. In Abbildung 3 wird das Ergebnis des besten Modells visualisiert.

	Model	MAE	MSE	RMSE	R2	RMSLE	MAPE	TT (Sec)
ada	AdaBoost Regressor	125.5175	27051.3514	140.3037	-41.4762	0.5884	0.8558	0.0260
knn	K Neighbors Regressor	124.9048	27447.3526	140.7198	-40.1014	0.5955	0.8762	0.0140
rf	Random Forest Regressor	126.0251	27786.5837	143.0590	-46.2983	0.6068	0.8876	0.0610
catboost	CatBoost Regressor	126.9630	27919.9770	144.1081	-51.4384	0.6122	0.8960	0.7190
gbr	Gradient Boosting Regressor	127.3030	28084.0812	144.8509	-53.1830	0.6138	0.8969	0.0160
et	Extra Trees Regressor	127.8359	28118.4634	144.4703	-50.2567	0.6132	0.8982	0.0610
xgboost	Extreme Gradient Boosting	132.1271	29953.6669	149.0889	-60.0022	0.6249	0.9119	0.0650
dt	Decision Tree Regressor	133.9091	30607.1030	150.6653	-61.7761	0.6294	0.9181	0.0100
lightgbm	Light Gradient Boosting Machine	164.7740	38410.6059	178.5892	-120.1722	0.7217	1.1194	0.0170
llar	Lasso Least Angle Regression	245.8038	132348.2289	269.4620	-484.6923	0.8431	1.4576	0.0170
ard	Automatic Relevance Determination	262.5664	163004.6556	288.2844	-646.5040	0.8491	1.4919	0.0180
br	Bayesian Ridge	268.7561	164105.6442	294.2654	-649.1822	0.8688	1.5217	0.0270
lr	Linear Regression	271.0957	165567.7801	296.6424	-661.0603	0.8741	1.5319	0.2220
omp	Orthogonal Matching Pursuit	271.0957	165567.8069	296.6424	-661.0601	0.8741	1.5319	0.0180
ridge	Ridge Regression	271.0957	165567.7543	296.6424	-661.0599	0.8741	1.5319	0.0110
lasso	Lasso Regression	271.0951	165563.6863	296.6414	-661.0525	0.8741	1.5319	0.0090
en	Elastic Net	271.0951	165564.7685	296.6415	-661.0561	0.8741	1.5319	0.0110
lar	Least Angle Regression	271.0957	165567.8069	296.6424	-661.0601	0.8741	1.5319	0.0100
huber	Huber Regressor	278.9318	181180.5390	304.1119	-957.6683	0.8757	1.5380	0.0160
svm	Support Vector Regression	161.2965	45615.2225	170.1008	-174.2990	0.7778	1.5733	0.0200
mlp	MLP Regressor	355.1865	315681.6709	386.2346	-2161.6519	0.9380	1.5851	0.0420
dummy	Dummy Regressor	168.9720	45542.4178	177.0765	-255.8877	0.8090	1.7568	0.0140
ransac	Random Sample Consensus	380.4167	305830.5425	404.0496	-1472.0791	1.0365	2.0499	0.0170
tr	TheilSen Regressor	709.5288	1269189.0097	778.3091	-8520.5650	1.2155	3.5109	0.1570
par	Passive Aggressive Regressor	1504.3282	6592165.6479	1655.7620	-20940.2815	1.5448	6.0069	0.0190
kr	Kernel Ridge	93895.2354	161305907942.6182	129458.5100	-392526142.3169	2.0438	441.1370	0.0180

Abbildung 2: Output von `compare_models` mit `pycaret.regression`

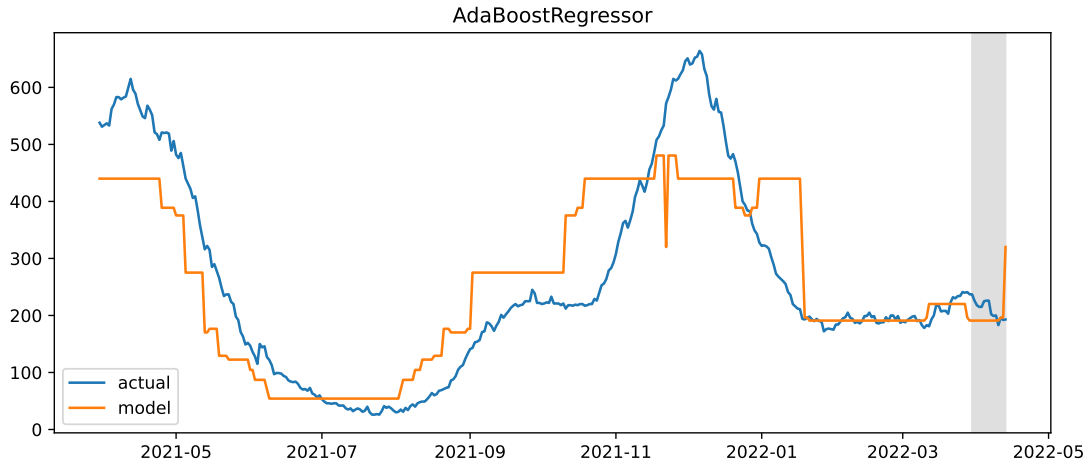

Abbildung 3: Das beste Modell aus Abbildung 2

Wie man leicht erkennt, liefern die Regressionsmodelle keine zufriedenstellenden Ergebnisse. Das schlechte Ergebnis hat auch nicht mit dem gewählten Zeitraum zu tun: es sah bei allen getesteten Zeiträumen sehr ähnlich aus; nur die „besten“ Modelle haben sich häufig geändert. Scheinbar sind also die normalen Regressionsmodelle nicht für unsere Problemstellung geeignet. Deshalb wurden im Anschluss die Modelle aus `pycaret.timeseries` getestet, welche (wie der Name sagt) speziell auf Zeitreihen ausgelegt sind. Dabei wurden die geglätteten Fallzahlen als exogene Daten verwendet. Es wurden wieder dieselben Tests wie mit den Regressionsmodellen durchgeführt. Das Ergebnis des Leistungsvergleichs ist in Abbildung 5 zu sehen. Das beste Modelle aus diesem Vergleich ist in Abbildung 4 visualisiert.

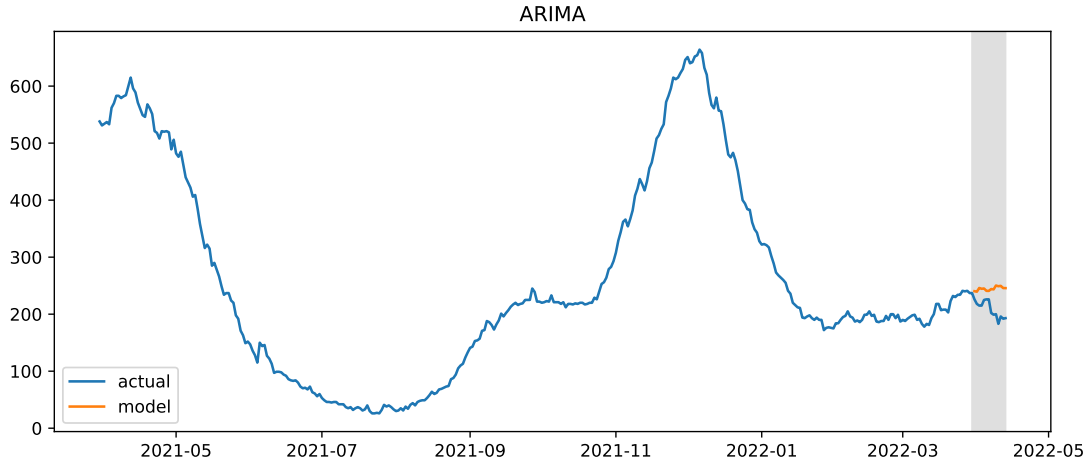

Abbildung 4: Das beste Modell aus Abbildung 5

	Model	MASE	RMSSE	MAE	RMSE	MAPE	SMAPE	R2	TT (Sec)
arima	ARIMA	0.9074	0.8188	33.4638	39.2007	0.1067	0.1062	-7.6897	0.1310
auto_arima	Auto ARIMA	0.9933	0.8940	36.5249	42.8484	0.1116	0.1093	-6.7584	10.9230
omp_cds_dt	Orthogonal Matching Pursuit w/ Cond. Deseasonalize & Detrending	1.1133	0.9716	40.4648	46.0660	0.1189	0.1123	-2.7929	0.1450
naive	Naive Forecaster	1.1932	1.0330	43.3571	48.9694	0.1244	0.1156	-2.9827	0.3500
lr_cds_dt	Linear w/ Cond. Deseasonalize & Detrending	1.2385	1.1426	46.5870	56.0356	0.1867	0.1579	-25.1693	0.1910
ridge_cds_dt	Ridge w/ Cond. Deseasonalize & Detrending	1.2386	1.1426	46.5884	56.0370	0.1867	0.1579	-25.1702	0.1410
br_cds_dt	Bayesian Ridge w/ Cond. Deseasonalize & Detrending	1.2722	1.1682	47.7901	57.2330	0.1907	0.1607	-26.0104	0.1480
en_cds_dt	Elastic Net w/ Cond. Deseasonalize & Detrending	1.3186	1.1928	49.5113	58.4031	0.1958	0.1589	-31.4400	0.1500
lasso_cds_dt	Lasso w/ Cond. Deseasonalize & Detrending	1.3217	1.1952	49.6102	58.4994	0.1960	0.1590	-31.4175	0.1600
xgboost_cds_dt	Extreme Gradient Boosting w/ Cond. Deseasonalize & Detrending	1.3557	1.2150	49.9018	58.3165	0.1608	0.1413	-6.8474	0.2540
huber_cds_dt	Huber w/ Cond. Deseasonalize & Detrending	1.3927	1.2672	51.3227	61.1392	0.1879	0.1723	-30.8738	0.1560
et_cds_dt	Extra Trees w/ Cond. Deseasonalize & Detrending	1.4161	1.2582	51.7769	60.1008	0.1611	0.1426	-9.8184	0.3030
ada_cds_dt	AdaBoost w/ Cond. Deseasonalize & Detrending	1.4524	1.2443	52.1363	58.5148	0.1421	0.1364	-7.6730	0.1720
gbr_cds_dt	Gradient Boosting w/ Cond. Deseasonalize & Detrending	1.4887	1.2919	54.2160	61.4954	0.1588	0.1436	-7.3287	0.1920
rf_cds_dt	Random Forest w/ Cond. Deseasonalize & Detrending	1.5370	1.3325	55.8417	63.3260	0.1619	0.1467	-8.3908	0.3170
dt_cds_dt	Decision Tree w/ Cond. Deseasonalize & Detrending	1.5417	1.3351	55.5973	62.9953	0.1532	0.1464	-12.5881	0.1290
catboost_cds_dt	CatBoost Regressor w/ Cond. Deseasonalize & Detrending	1.6234	1.3897	59.6660	66.5688	0.1871	0.1650	-10.4415	2.5180
snaive	Seasonal Naive Forecaster	1.7539	1.4457	63.5857	68.5591	0.1827	0.1715	-9.2297	0.0680
lightgbm_cds_dt	Light Gradient Boosting w/ Cond. Deseasonalize & Detrending	1.7982	1.5205	65.5605	72.5213	0.2047	0.1858	-16.8613	0.1450
llar_cds_dt	Lasso Least Angular Regressor w/ Cond. Deseasonalize & Detrending	2.1750	1.8330	77.1317	85.2906	0.2177	0.2243	-43.8428	0.1270
croston	Croston	2.3590	1.8885	85.9299	89.8421	0.2556	0.2308	-24.2310	0.0570
knn_cds_dt	K Neighbors w/ Cond. Deseasonalize & Detrending	3.1340	2.6110	119.2021	128.3219	0.4484	0.2956	-265.2241	0.1790
grand_means	Grand Means Forecaster	3.9797	3.0994	141.8194	144.9740	0.3906	0.4281	-118.1788	0.0640
par_cds_dt	Passive Aggressive w/ Cond. Deseasonalize & Detrending	11.9671	11.1609	457.7724	552.2970	2.0531	0.8431	-14358.7531	0.1380

Abbildung 5: Output von `compare_models` mit `pycaret.timeseries`

Die Prognose in Abbildung 4 ist natürlich nicht perfekt, aber ein Blick auf den MAPE verrät, dass das ARIMA Modell viel besser als die Modelle aus `pycaret.regression` ist. Auch bei allen anderen getesteten Zeiträumen erwies sich ARIMA bzw. `auto_arima` als das beste Modell aus `pycaret.timeseries`. Aus diesem Grund verwenden wir für den datenbasierten Ansatz ein ARIMA Modell.

4.1 ARIMA Modell

Bevor wir die Implementierung des ARIMA Modells beschreiben, wollen wir uns zuerst die Theorie dahinter ansehen. Die Informationen dafür stammen von [Wik22]. ARMA(p,q) ist ein lineares, zeitdiskretes Modell für Zeitreihen. ARMA ist ein Akronym für **A**uto**R**egressive - **M**oving **A**verage. Sei y_t das zu modellierende Signal und ε_t eine Folge normalverteilter Fehlerterme (auch Rauschterme genannt) mit Erwartungswert 0. Bei einem *moving average* Modell MA(q) wird angenommen, dass sich das Signal y_t als gewichtetes gleitendes Mittel der q vorherigen Fehlerterme schreiben lässt, d.h.

$$y_t = c + \epsilon_t + \sum_{i=1}^{q} \beta_i \varepsilon_{t-i}.$$

Bei einem *autoregressive* Modell AR(p) setzt sich das Signal y_t aus dem gleitenden Mittel der p vorhergehenden Signalwerte zusammen, d.h.

$$y_t = c + \varepsilon_t + \sum_{i=1}^{q} \alpha_i y_{t-i}.$$

Ein ARMA(p,q) Modell ist schlicht die Kombination eines AR(p) Modells mit einem MA(q) Modell. Bei einem ARMA(p,q) nimmt man also

$$y_t = c + \varepsilon_t + \sum_{i=1}^{q} \alpha_i y_{t-i} + \sum_{i=1}^{q} \beta_i \varepsilon_{t-i}$$

für das zu modellierende Signal y_t an. ARMA(p,q) ist eine echte Verallgemeinerung von *moving average* und *autoregressive* Modellen: setzt man $p = 0$ erhält man MA(q); setzt man $q = 0$ erhält man AR(p).

ARIMA(p,d,q) ist eine Erweiterung von ARMA(p,q), wobei das I für „**I**ntegrated“ steht. Ein ARMA(p,q) Modell ist stationär, und macht somit nur Sinn, wenn auch y_t stationär ist. Falls y_t nicht stationär ist, kann häufig durch wiederholte Differenzenbildung $y_t - y_{t-1}$ Stationarität erzeugt werden. Der Paramter d eines ARIMA(p,d,q) Modells gibt an, wie oft das Eingangssignal y_t vor dem Einsetzen in das ARMA(p,q) Modell differenziert wird. Das Integrieren ist dann die Umkehroperation, um wieder das undifferenzierte Signal zu erhalten.

ARIMAX ist eine Erweiterung von ARIMA, welche es ermöglicht exogene (auf Englisch „e**X**ogenous“) Variablen bei der Modellierung des Signals y_t zu verwenden. Seien $(x_{n_t})_{n=1}^{N}$ exogene Variablen. Dann unterscheidet sich die Gleichung eines ARIMAX(p,d,q) Modells von einem ARIMA(p,d,q) Modell nur um eine weitere Summe, nämlich

$$y_t = c + \varepsilon_t + \sum_{i=1}^{q} \alpha_i y_{t-i} + \sum_{i=1}^{q} \beta_i \varepsilon_{t-i} + \sum_{n=1}^{N} \gamma_n x_{n_t}.$$

Setzen wir $x_t \equiv 0$, so erhalten wir wieder ein ARIMA Modell.

SARIMAX ist wiederum eine Erweiterung von ARIMAX, welche es ermöglicht, saisonale Effekte zu berücksichtigen (daher auch „**S**easonal“). Dies wird erreicht, indem eine zusätzliche ARMA Komponente mit Parametern P, D und Q zur Gleichung hinzugefügt wird: angenommen das Signal y_t hat eine saisonale Periode m, dann ist

$$y_t = c + \varepsilon_t + \sum_{i=1}^{q} \alpha_i y_{t-i} + \sum_{i=1}^{q} \beta_i \varepsilon_{t-i} + \sum_{n=1}^{N} \gamma_n x_{n_t} + \sum_{i=1}^{Q} \phi_i y_{t-mi} + \sum_{i=1}^{P} \theta_i \varepsilon_{t-mi}$$

die Gleichung des SARIMAX Modells mit Parametern $(p, d, q) \times (P, D, Q)_m$. Setzt man $P = D = Q = 0$ erhält man wieder ein ARIMAX Modell.

Die Gewichte $\alpha_i, \beta_i, \gamma_n, \phi_i$ und θ_i können mittels Maximum-Likelihood-Schätzung oder Kleinste-Quadrate-Schätzung bestimmt werden. Wesentlich interessanter ist die Frage, wie die Parameter p, d, q, P, D und Q bestimmt werden. Dafür werden im Normalfall Kriterien zur Modellselektion[7] wie das Akaike-Informationskriterium (AIC) verwendet;

[7] `https://de.wikipedia.org/wiki/Informationskriterium`

je niedriger das AIC, desto besser die Parameter. Daher ist es eine legitime Strategie, alle möglichen Kombinationen für p, d, q, P, D und Q auszuprobieren, und jenes Modell mit dem niedrigsten AIC auszuwählen.

Genau das macht die Funktion `auto_arima` aus dem Python package `pmdarima`. Wie schon bei den `PyCaret` Tests, verwenden wir die geglätteten Fallzahlen als exogene Variable. Da `PyCaret` eine Saisonalität mit einer Periodelänge von 7 Tagen festgestellt hat, setzen wir $m = 7$. Weil diese Saisonalität in den Hospitalisierungen visuell nicht wirklich ausgeprägt ist, stellt sich die Frage, ob dieser Schritt wirklich notwendig ist, oder vielleicht doch ein ARIMAX Modell ausreichen würde. Wenn man allerdings bedenkt, dass ein SARIMAX Modell mit $P = D = Q = 0$ genau ein ARIMAX Modell ist, stört das Berücksichtigen einer (eventuellen) Saisonalität nicht. Da nach dem Augmented Dickey–Fuller Test[8] die Fallzahlen bereits nach einmaligem Differenzieren stationär sind, beschränken wir d und D durch 1 nach oben. Für die anderen Parameter haben wir folgende Grenzen festgelegt:

$$p, q \in \{0, \dots, 7\}, \quad P, Q \in \{0, \dots, 2\}.$$

Die Grenzen wurden eher willkürlich festgelegt, sind aber noch immer großzügiger als die *defaults* von `auto_arima` (was rückblickend nicht notwendig gewesen wäre, siehe Kapitel 5). Um die Nutzung dieses Modells für das Projekt zu vereinfachen, wurde die `ArimaModel` Klasse geschrieben. Ihre Verwendung ist im Quellcode dokumentiert.

4.2 Laufzeit

Für dasselbe Setting wie in 3.4 hat die Kalibierung des ARIMA Modells eine Laufzeit von 2 Sekunden. Die Prognose der nächsten 14 Tage braucht 36 ms.

5 Evaluation

Um die Prognoseleistung der Modelle zu evaluieren, wurde folgendes Skript verwendet:

Code 1: `benchmark.py`

```
def benchmark(forecast: callable, dates: list, forecast_days: int,
              state: str, bed_type: str, occupancy_data: OccupancyData,
              path="benchmark.json"):
    successful = dict()
    failed = dict()
    for date in dates:
        try:
            prediction, calibration = forecast(date, forecast_days)
            from_date = pd.to_datetime(date) + pd.to_timedelta(1, unit="d")
            to_date = from_date + pd.to_timedelta(forecast_days-1, unit="d")
            reference = occupancy_data.get_array(from_date, to_date,
                                                 state, bed_type)
```

[8] https://en.wikipedia.org/wiki/Augmented_Dickey%E2%80%93Fuller_test

```
            successful[date] = dict()
            successful[date]["actual"] = to_list(reference)
            successful[date]["prediction"] = to_list(prediction)
            successful[date]["diff"] = to_list(reference - prediction)
            successful[date]["mase"] = mase(reference, prediction)
            successful[date]["mape"] = mape(reference, prediction)
            successful[date]["mae"] = mae(reference, prediction)
            successful[date]["calibration_stats"] = calibration
            print(f"Forecast MASE for {date}: {mase(reference, prediction)}")
        except Exception as e:
            print(f"Forecast for {date} failed: {e}")
            failed[date] = str(e)
    result = {
        "forecast_days": forecast_days,
        "state": state,
        "bed_type": bed_type,
        "successful_forecasts": successful,
        "failed_forecasts": failed,
        "dates": dates
    }
    with open(path, "w") as f:
        json.dump(result, f, cls=MyEncoder, indent=4, ensure_ascii=False)
```

Das Skript funktioniert mit jeder Funktion `forecast`, die für ein gegebenes Datum eine Prognose zurückliefern kann. Die Idee besteht darin, dass mit dem Parameter `dates` eine Liste von Daten übergeben werden kann, an denen eine Prognose erstellt werden soll. Wie weit in die Zukunft vorhergesagt werden soll, wird über den Parameter `forecast_days` geregelt. Wir haben uns in unseren Tests für 14 Tage entschieden, da das auch die Länge der Prognosen des COVID-Progonose-Konsortiums[9] ist. Nach dem Erstellen der Prognose wird diese mit den tatsächlichen Werten verglichen und bestimmte Fehlermaße berechnet. Neben dem MAPE berechnen wir dabei auch den MAE und den MASE. Der *mean absolute error* (MAE) ist (wie der Name schon sagt) der Mittelwert der absoluten Fehler:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|$$

Der *mean absolute scaled error* (MASE) ist ein Maß für die Genauigkeit von Prognosen, welcher gewisse Nachteile des MAE und MAPE ausbessert, und allgemein besser zum Vergleich der Prognosegenauigkeit zwischen verschiedenen Zeitreihen geeignet sein soll[10]. Der MASE von x_i bezüglich y_i ist folgendermaßen definiert:

$$MASE = \frac{\frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|}{\frac{1}{n-1}\sum_{i=2}^{n}|y_i - y_{i-1}|}$$

Die Ergebnisse werden dann in einer JSON-Datei gespeichert, was eine spätere Analyse der Modelle ermöglicht. Wir werden das Benchmark-Skript verwenden, um die optimale

[9] `https://www.sozialministerium.at/Corona/zahlen-daten/COVID-Prognose-Konsortium-2022.html`

[10] `https://robjhyndman.com/papers/foresight.pdf`

Länge des Kalibrierungsintervalls zu bestimmen und dann final zu entscheiden, welcher Modellierungsansatz der bessere für unser Problem ist.

5.1 Bestimmtung des Kalibrierungsintervalls

Zunächst wollen wir das optimale Kalibrierungsintervall für das kausale Modell bestimmen. Wir testen dafür mittels des Benchmark-Skripts Kalibrierungsintervalle der Länge 7, 14, 21, 28, 35, 42, 49 und 56 Tage. Für die Daten, an denen die Prognose stattfinden soll, haben wir 24 äquidistante Tage im Zeitraum von 08.12.2020 bis 16.09.2022 gewählt (Abbildung 6):

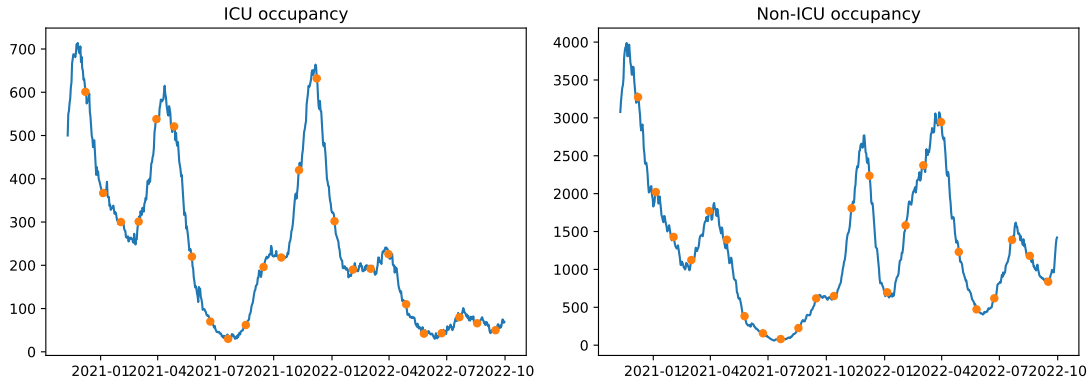

Abbildung 6: Daten für die Bestimmung des Kalibrierungsintervalls

Die Ergebnisse sind in Abbildung 7 und Tabelle 1 zusammengefasst.

Intervall (in Tagen)	**Mittelwert MASE**	**Mittelwert MAPE**	**Std. Abw. MASE**	**Std. Abw. MAPE**	**Kalibr. MAPE**
7	3.504236	13.581280	3.153912	10.498583	2.128433
14	3.813773	14.638348	3.285414	11.129703	3.325302
21	5.710411	18.969642	9.624056	24.223855	4.092146
28	7.877509	25.649568	12.086507	29.337454	4.646206
35	9.256486	29.987352	13.817893	33.129465	5.719202
42	9.781833	31.315156	14.229476	33.419280	6.562721
49	11.297007	35.853156	17.994700	42.907354	7.241986
56	11.412634	36.524942	18.210043	43.664305	8.484697

Tabelle 1: Test der Kalibrierungsintervalle 7, 14, 21, 28, 35, 42, 49 und 56 Tage

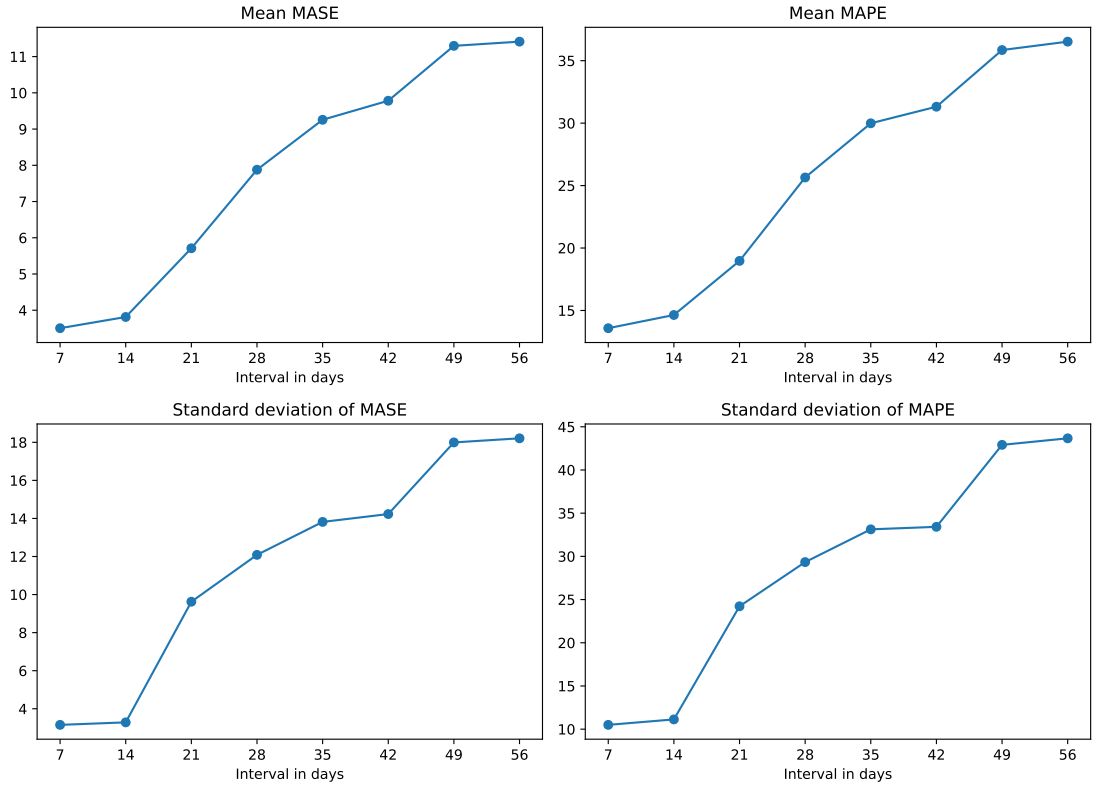

Abbildung 7: Visualisierung der Ergebnisse aus Tabelle 1

Man erkennt eindeutig, dass das 7 tägige Kalibrierungsintervall mit Abstand die besten Ergebnisse geliefert hat, und dass die Fehler mit größeren Kalibrierungsintervallen zugenommen haben. Das erscheint im ersten Moment etwas unintuitiv; 7 Tage sind schon sehr kurz. Allerdings erkennt man in Tabelle 1 auch den Grund dafür: Je länger das Kalibrierungsintervall war, desto schlechter waren die Modelle im Schnitt kalibriert. Das macht auch Sinn: im echten Leben sind die Parameter, anders als im Modell, sicherlich nicht konstant, und je länger das Intervall ist, umso mehr können sich die Parameter im Zeitraum ändern. Die stark zunehmende Standardabweichung spricht allerdings dafür, dass längere Kalibrierungsintervalle nicht universell schlechter sind, sondern lediglich bei einigen Ausreißern besonders schlecht abgeschnitten haben. Diese Theorie wird durch durch Abbildung 8 bestätigt.

Aus diesem Grund würde es eigentlich Sinn machen, das Kalibrierungsintervall individuell auf den jeweiligen Prognosezeitraum anzupassen. Da wir uns allerdings in diesem Projekt eine *one-fits-all* Lösung wünschen, wiederholen wir den obigen Test mit Intervalllängen im Bereich von 7 Tagen, um eventuell ein noch besseres Kalibrierungsintervall zu finden. Die Ergebnisse sind in Abbildung 9 und Tabelle 2 zusammengefasst.

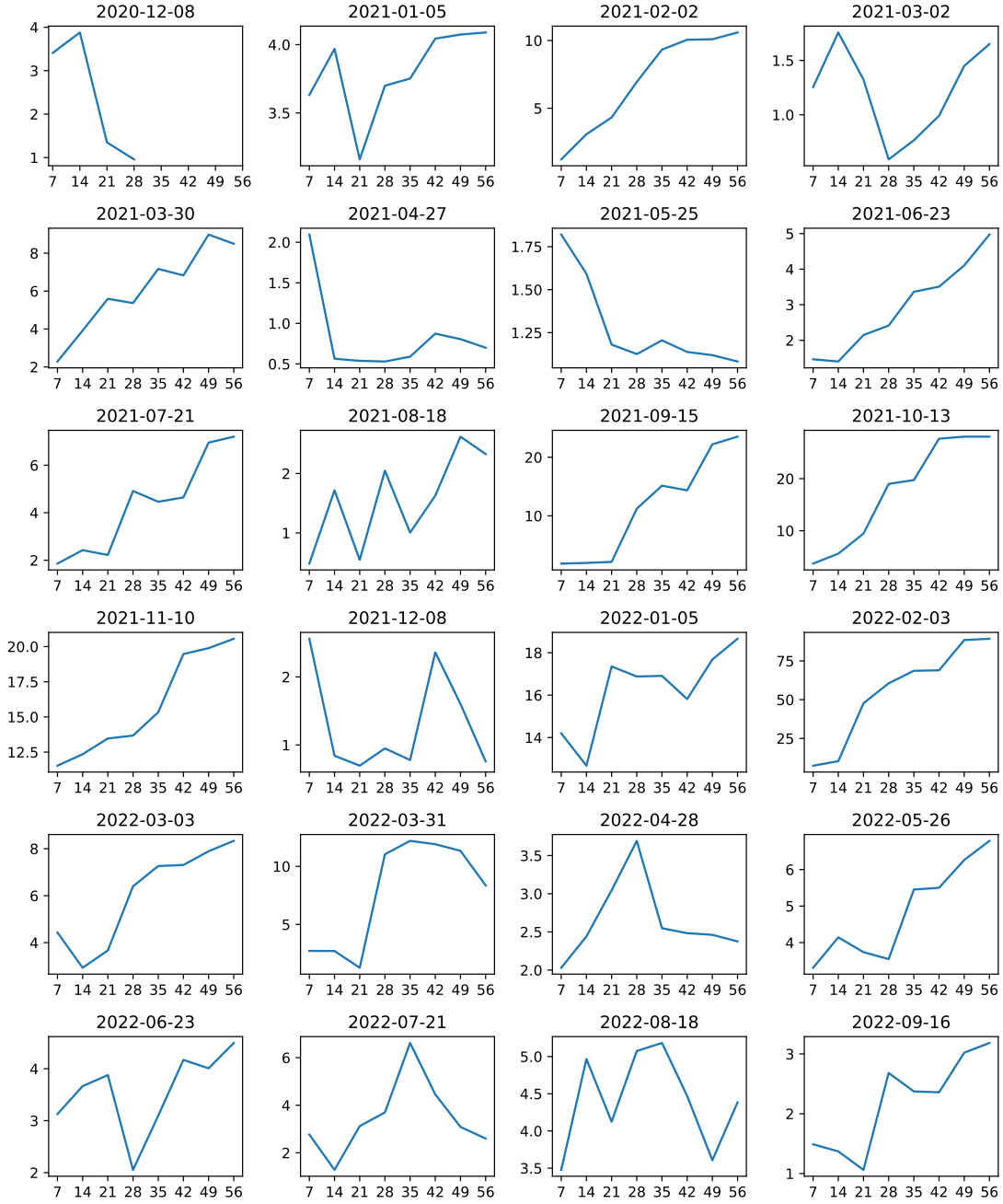

Abbildung 8: Visualisierung des MASE für jedes der 24 Daten

Intervall (in Tagen)	**Mittelwert MASE**	**Mittelwert MAPE**	**Std. Abw. MASE**	**Std. Abw. MAPE**	**Kalibr. MAPE**
3	3.651953	15.038056	2.654208	10.726763	1.462746
4	3.985951	16.149921	4.121159	14.414733	1.522998
5	3.493775	14.072875	2.918518	10.098462	1.784223
6	3.333530	13.375115	2.681120	10.049814	1.917913
7	3.504236	13.581280	3.153912	10.498583	2.128433
8	3.585203	14.177686	3.425326	11.962872	2.358760
9	3.565755	14.204601	3.263180	11.662252	2.547729
10	3.585102	13.849740	3.424440	11.930181	2.892161
11	3.639011	14.243394	3.444444	11.996330	2.912200

Tabelle 2: Test der Kalibrierungsintervalle 3, 4, 5, 6, 7, 8, 9, 10 und 11 Tage

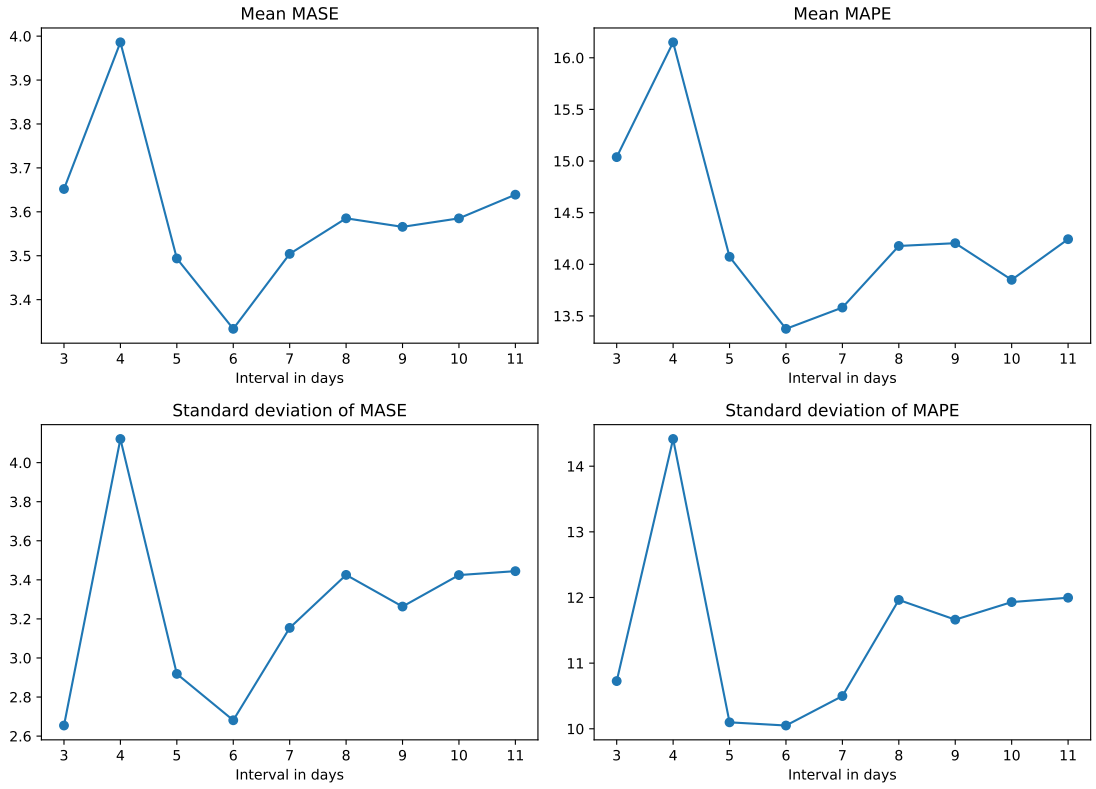

Abbildung 9: Visualisierung der Ergebnisse aus Tabelle 2

Die Ergebnisse aus Abbildung 9 sprechen eindeutig für ein 6 Tage langes Kalibrierungsintervall.

Dieselben Tests haben wir auch für das ARIMA Modell durchgeführt. Aufgrund der kürzeren Laufzeit des ARIMA Modells konnten wir wesentlich mehr Intervalllängen testen, unter anderem auch die jeweils maximal mögliche Intervalllänge. Die Ergebnisse sind in Abbildung 10 und Tabelle 3 zusammengefasst.

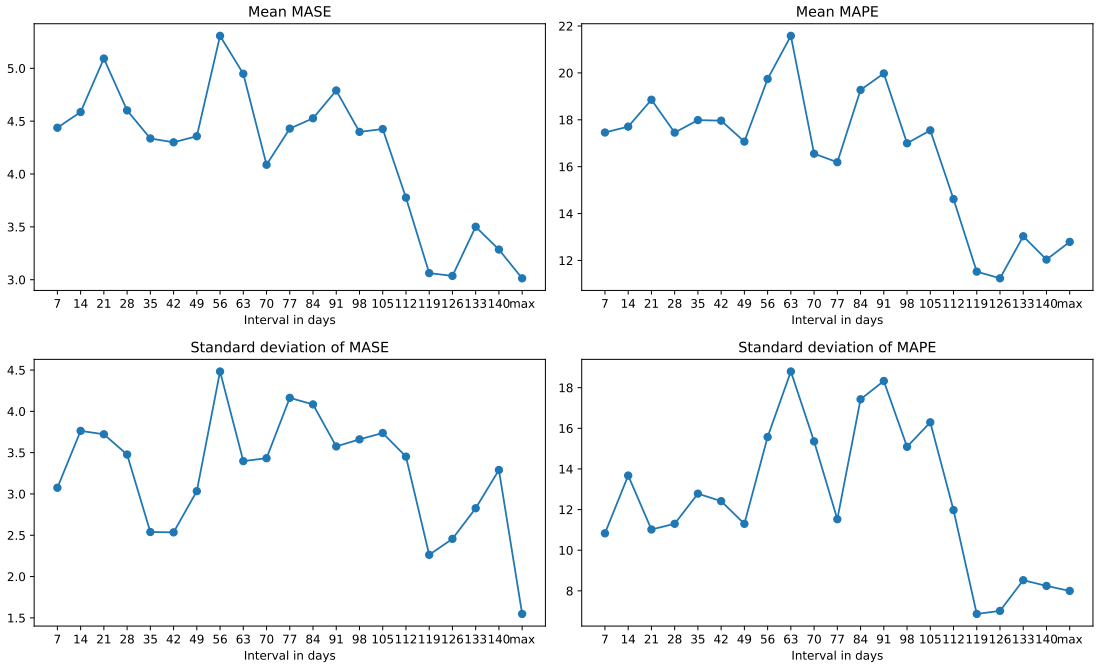

Abbildung 10: Visualisierung der Ergebnisse aus Tabelle 3

Intervall (in Tagen)	**Mittelwert MASE**	**Mittelwert MAPE**	**Std. Abw. MASE**	**Std. Abw. MAPE**
7	4.437975	17.457967	3.075183	10.832493
14	4.586590	17.707964	3.763605	13.674937
21	5.092651	18.855563	3.722407	11.020240
28	4.601896	17.453824	3.477197	11.300565
35	4.336234	17.986894	2.539206	12.784802
42	4.300025	17.962562	2.535113	12.414851
49	4.357621	17.069763	3.034267	11.299231
56	5.307134	19.741674	4.482483	15.572121
63	4.948208	21.580033	3.397474	18.796575
70	4.087460	16.554287	3.432612	15.356607
77	4.429801	16.190675	4.163375	11.525899
84	4.527095	19.273205	4.083908	17.429743
91	4.790337	19.977723	3.575227	18.328709

98	4.398734	16.998286	3.660795	15.090777
105	4.425621	17.549795	3.737860	16.295220
112	3.776579	14.617364	3.451853	11.974182
119	3.062558	11.523723	2.263135	6.863339
126	3.036030	11.238481	2.456385	7.013613
133	3.501193	13.032380	2.826884	8.524184
140	3.285466	12.037620	3.290097	8.243636
max	3.013221	12.791892	1.547390	7.997386

Tabelle 3: Test verschiedener Kalibrierungsintervalle für das ARIMA Modell

Aufgrund der Ergebnisse in Abbildung 10 haben wir uns entschieden, für das ARIMA Modell stets das maximal mögliche Intervall zur Kalibrierung zu verwenden.

5.2 Kausales Modell vs. ARIMA Modell

Da nun die Kalibrierungsintervalle feststehen, können wir jetzt testen, welches der Modelle bessere Prognosen liefert. Dafür verwenden wir das Benchmark-Skript wie bei der Intervallfindung, mit dem einzigen Unterschied, dass wir anstelle von 24 Daten nun 150 Tage testen. Diese sind äquidistant über den Bereich von 15.12.2020 bis 15.09.2022 verteilt:

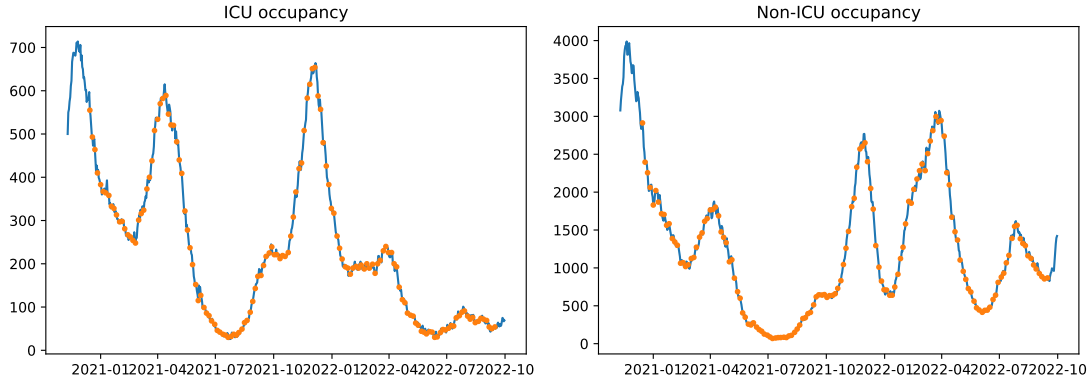

Abbildung 11: Daten für den Vergleich zwischen kausalem Modell und ARIMA Modell

Die Ergebnisse der beiden Tests ist in den nachfolgenden Tabellen 4 und 5 zusammengefasst:

Mean MASE	`4.1707882844294835`
Mean MAPE	`15.40278438879675`
Standard deviation MASE	`5.928188310942641`
Standard deviation MAPE	`17.422097426035535`

Mean calibration MAPE	`1.9897156112593413`

Tabelle 4: Evaluation: Kausales Modell

Mean MASE	`3.8787611627728515`
Mean MAPE	`18.25590051780181`
Standard deviation MASE	`3.1282478456386915`
Standard deviation MAPE	`24.279225452055428`

Tabelle 5: Evaluation: ARIMA Modell

Zwar hat das ARIMA Modell im Schnitt einen niedrigeren MASE als das kausale Modell; bezüglich des MAPE schneidet aber das kausale Modell besser ab. Somit lässt sich die Frage, welcher Modellierungsansatz in diesem Projekt bessere Ergebnisse liefert, nicht eindeutig beantworten: die Antwort hängt von dem Fehlermaß ab.

Eine interessante Beobachtung zeigt folgende Grafik, welche die Häufigkeit der in der Evaluation aufgetretenen Parameterkombinationen $(p, d, q) \times (P, D, Q)$ vergleicht:

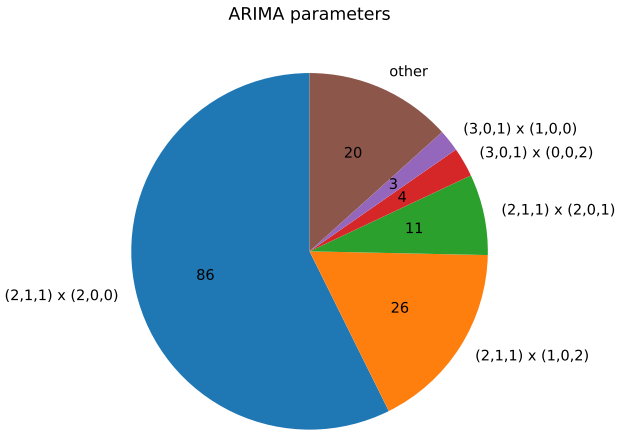

Abbildung 12: Parameterkombinationen des ARIMA Modells in der Evaluation

Man erkennt, dass sehr wenig verschiedene SARIMAX$(p, d, q) \times (P, D, Q)$ Modelle verwendet wurden.

6 Konfidenzintervalle

In diesem Abschnitt wollen wir Methoden entwickeln, um Konfidenzintervalle für die implementierten Modelle angeben zu können. Bei dem ARIMA Modell gestaltet sich

diese Aufgabe sehr einfach: Da ARIMA ein sehr populäres statistisches Modell ist, gibt es natürlich bereits etablierte Methoden zur Bestimmung des Konfidenzintervalls. Eine solche Methode ist auch in `pmdarima` implementiert; indem man in der `predict` Methode `return_conf_int=True` setzt, erhält man (zusammen mit der Prognose) ein Konfidenzintervall für die Prognose. Die dafür verwendete Methode ist allerdings nicht näher beschrieben, außer dass sie auf Normalverteilungsannahmen basiert. Auch im Allgemeinen würde die Beschreibung einer solchen Methode den Rahmen dieser Dokumentation sprengen, da dafür sehr viel Theorie benötigt wird.

Wir widmen uns daher nun der Berechnung von Konfidenzintverallen für das kausale Modell. Da wir aus der Evaluation in Abschnitt 5.2 sehr viele Daten über den Prognosefehler des Modells haben, wurde folgender Ansatz gewählt: Sei $y_1, \dots, y_{14}$ eine Prognose für $x_1, \dots, x_{14}$. Wir definieren e_i als den relativen Unterschied von x_i bezüglich y_i, d.h.

$$e_i := \frac{x_i - y_i}{y_i}$$

für alle $i \in \{1, \dots, 14\}$. Somit gilt

$$x_i = y_i + e_i \cdot y_i, \quad i \in \{1, \dots, 14\}. \tag{$*$}$$

Unter der Annahme, dass e_i für alle Prognosen die gleiche Verteilung F_i hat und F_i bekannt ist, sollte x_i nach $(*)$ mit 95%-iger Wahrscheinlichkeit im Intervall

$$[y_i + q_{0{,}025} \cdot y_i, y_i + q_{0{,}975} \cdot y_i]$$

liegen, wobei q_p das p-Quantil von F_i ist. Mit den Daten aus 5.2 haben wir eine große Stichprobe ($n = 150$) von e_i, $i = 1, \dots, 14$. Mit dieser Stichprobe können wir mittels gaußschem Kerndichteschätzer[11] die Dichtefunktion von F_i approximieren und die Quantile berechnen. Dafür verwenden wir `gaussian_kde` aus `scipy.stats`. In Tabelle 6 sehen wir die damit berechneten Quantile für die e_i. In Abbildung 13 sind die Histogramme der Stichprobe von e_1, e_7 und e_{14} zusammen mit der approximierten Dichtefunktion dargestellt.

	$q_{0{,}025}$	$q_{0{,}975}$
e_1	-0.18	0.21
e_2	-0.23	0.27
e_3	-0.24	0.28
e_4	-0.29	0.28
e_5	-0.36	0.35
e_6	-0.36	0.40
e_7	-0.40	0.42
e_8	-0.44	0.44

[11] https://de.wikipedia.org/wiki/Kerndichtesch%C3%A4tzer

e_9	-0.47	0.46
e_{10}	-0.50	0.50
e_{11}	-0.52	0.49
e_{12}	-0.56	0.53
e_{13}	-0.57	0.59
e_{14}	-0.59	0.57

Tabelle 6: Quantile für das kausale Modell

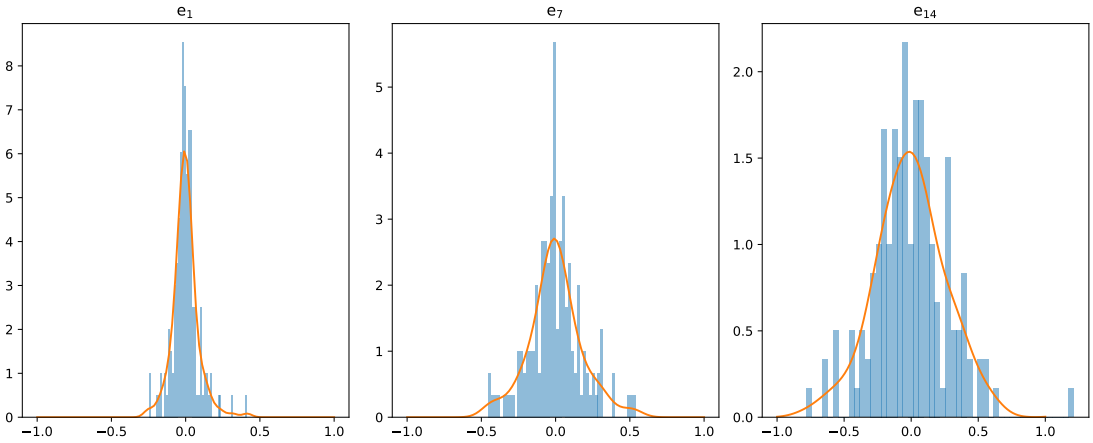

Abbildung 13: Approximation der Dichtefunktion von e_1, e_7 und e_{14}

In der nachfolgenden Abbildung wurden die oben berechneten Quantile und $(*)$ verwendet, um ein Konfidenzintervall für eine Prognose aufzustellen:

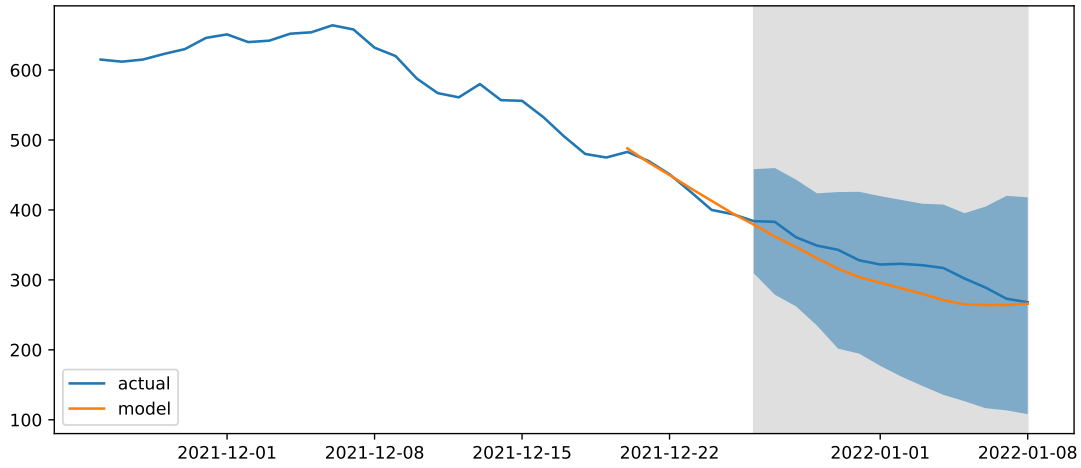

Abbildung 14: Kausales Modell: Prognose mit 95% Prognoseintervall

Wenn man es allerdings genau nimmt ist das, was wir oben berechnet haben, kein Konfidenzintervall, sondern ein Prognoseintervall: Konfidenzintervalle beziehen sich nämlich immer auf einen oder mehrere Parameter, die geschätzt werden[12]. Beim vorimplementierten Konfidenzintervall des ARIMA Modells sind das die Modellparameter α_i, β_i, usw. Wir können die obige Methode zur Erstellung eines Prognoseintervalls natürlich auch mit dem ARIMA Modell verwenden. In Abbildung 15 sind sowohl Konfidenz- als auch Prognoseintervall eingezeichnet. Man erkennt sehr gut, dass das Prognoseintervall wesentlich breiter als das Konfidenzintervall ist (was auch normalerweise der Fall ist).

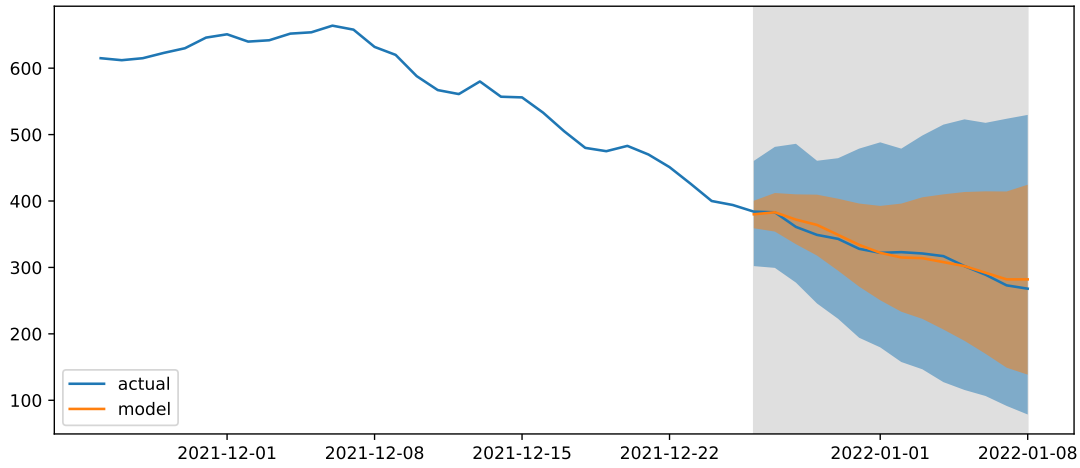

Abbildung 15: ARIMA Modell: Prognose mit 95% Prognoseintervall (blau) und 95% Konfidenzintervall (orange)

Literatur

[BRZ] Florian Bachner, Lukas Rainer, and Martin Zuba. Intensivpflege und COVID. Factsheet. `https://goeg.at/sites/goeg.at/files/inline-files/Factsheet%20COVID%20Hospitalisierungen_2022-09.pdf`. [zuletzt aufgerufen am 30.09.2022].

[RNJ+20] Eleanor M Rees, Emily S Nightingale, Yalda Jafari, Naomi R Waterlow, Samuel Clifford, Carl A B Pearson, Thibaut Jombart, Simon R Procter, Gwenan M Knight, et al. Covid-19 length of hospital stay: a systematic review and data synthesis. *BMC medicine*, 18(1):1–22, 2020.

[Wik22] Wikipedia. ARMA-Modell — Wikipedia. `http://de.wikipedia.org/w/index.php?title=ARMA-Modell&oldid=225229983`, 2022. [zuletzt aufgerufen am 30.09.2022].

[12] `https://robjhyndman.com/hyndsight/intervals/`