https://github.com/GraceZhu322/Data-Science-Project/tree/main

Abstract

This report presents a comprehensive analysis of various time series forecasting models applied to COVID-19 trend data, with an objective to identify the most effective model for predicting future trends. Utilizing methodologies like ARIMA, Auto_ARIMA, LSTM, XGBoost, and Prophet models (both univariate and multivariate), the project involved extensive data preparation, parameter tuning, and model evaluation. Key findings revealed XGBoost as the best-performing model due to its robust handling of complex data, followed by LSTM and the Prophet models. Despite its superior RMSE, the Prophet (Multivariate) model was favored for its balance between resource efficiency and predictive accuracy. A significant aspect of the project was the collaborative learning process, where group members engaged in mutual knowledge sharing and problem-solving, enhancing the overall understanding and application of diverse analytical approaches.

Project and Data Overview

Project Overview

The task of predicting daily new COVID-19 cases in the world's top 10 GDP per capita countries is vital, as their management of the pandemic significantly influences global economic and health policies.

This analysis is backed by a robust dataset from Google Health COVID-19 Open Data Repository. It's one of the most comprehensive collections of up-to-date COVID-19-related information. This dataset is crucial for public health experts and policymakers, offering diverse data types to enhance understanding and response strategies for COVID-19 in economically influential nations.

Data Overview and Preprocessing

The dataset for predicting daily new COVID-19 cases in the top 10 GDP per capita countries is extensive, covering over 20,000 locations globally. After manually aggregating various datasets based on location and date, the data expands to 12,525,825 rows and 198 columns. Focusing on the top 10 GDP per capita countries further refines the dataset to 97,184 rows × 198 columns, providing a concentrated view of these economically significant regions.

In preparing this dataset for time series prediction, several preprocessing steps are undertaken to address common challenges. Firstly, to handle missing data, columns with more than 20% missing values are dropped. Cumulative columns are imputed under the logic that their numbers never decrease, while KNN imputation is applied for non-cumulative columns. Feature engineering plays a critical role, where autocorrelation and partial correlation analyses are used to add lag features and rolling window statistics. Additionally, datetime features and seasonal variations are incorporated to enhance the model's accuracy.

A key aspect of preprocessing involves detecting outliers, especially in critical variables like new deceased cases, new fully vaccinated individuals, and mobility in workplaces. This process involves manually tuning thresholds to identify anomalies, ensuring the data's integrity and reliability for subsequent predictive modeling.

I have chosen Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE) to evaluate our predictive models. RMSE effectively quantifies average prediction errors, highlighting larger discrepancies, while MAPE provides an intuitive percentage-based accuracy measure. Together, they offer a balanced assessment of our model's performance.

Methodology

ARIMA Model

1. **Data Preparation**

Initially, I excluded data from before March 1, 2020, recognizing that the global outbreak of COVID-19 was not fully underway prior to this date. I then chose to group the data by 10 countries, selecting the United States as the representative country for building the model.

A crucial aspect of my preparation involved confirming the stationarity of the dataset, a key assumption for the effective application of ARIMA models. After differencing the dataset once, the Augmented Dickey-Fuller (ADF) test yielded a p-value of 0.527. While not ideal, this value was sufficient for me to assume stationarity with d=1 for the ARIMA model, indicating a single differencing.

The ARIMA model is characterized by three principal parameters: p (autoregressive), d (differencing), and q (moving average). Based on the ADF test results, I set the 'd' parameter to 1. The 'p' and 'q' values were determined through an analysis of the autocorrelation function (ACF) and partial autocorrelation function (PACF) plots.

2. **Model Building**

The initial step in the training process involved plotting the Autocorrelation Function (ACF) and the Partial Autocorrelation Function (PACF). These plots were crucial for providing initial insights into suitable values for the 'p' (autoregressive) and 'q' (moving average) parameters of the ARIMA model. Based on the patterns observed in these plots, I chose to start with 7 for both 'p' and 'q'.

To refine the model's parameters, I implemented a grid search strategy, systematically exploring all combinations of 'p' and 'q', each ranging from 1 to 7. This exhaustive search was essential to identify the most effective combination of parameters, ensuring the model's optimal performance. The result of this grid search was the final selection of parameters: 6 for 'p', 1 for 'd' (differencing), and 7 for 'q'. This combination, (6,1,7), was determined to be the most suitable for accurately modeling the time series data. Although it would be most ideal to tune a model for each country, due to the time constraint, I arbitrarily used the (6,1,7) combination, developed from US dataset, to build ARMA models for the rest of the countries.

During the training phase, one of the primary issues was ensuring the model adequately captured the underlying patterns in the data without overfitting. To address this, I carefully monitored the model's performance on a validation set, adjusting the parameters as needed.

3. **Model Performance (see Auto Arima for evaluation)**

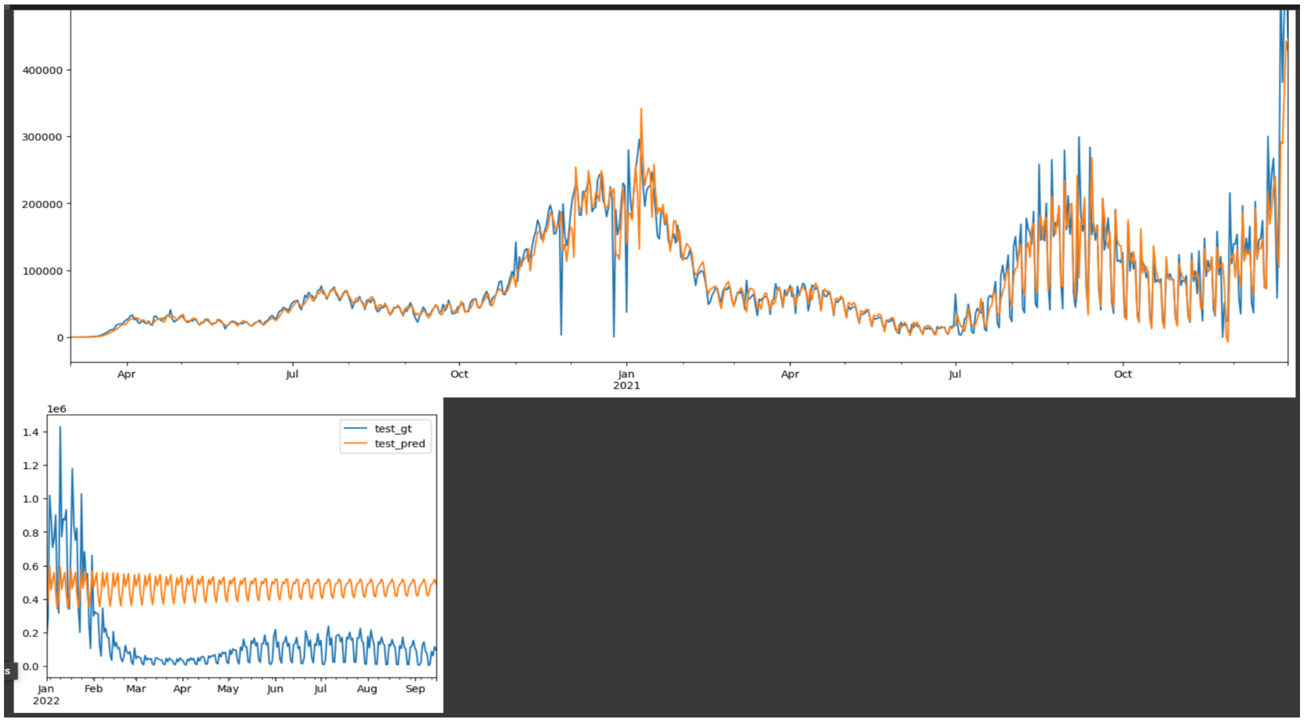
<u>Model focusing on the US:</u>

Train RMSE = 28633.74, MAPE = 445.55

Test RMSE = 382118.47, MAPE = 1159.07

<u>The entire dataset:</u>

Train RMSE = 9134.10, test RMSE = 123881.36



Auto_ARIMA Model

1. **Data Preparation**

In preparing the dataset for the Auto_ARIMA model, I largely followed the same steps as those used in the ARIMA model preparation. This consistency ensured that the data was suitably structured and cleansed for time series analysis. Similar to the ARIMA model, I focused on ensuring the stationarity of the dataset. I also grouped the data by counties and used US as a representative, allowing me to compare both the models for US and the model for the entire dataset to the previous ARIMA models.

The key distinction between Auto_ARIMA and the standard ARIMA model lies in the parameter selection process. While ARIMA requires manual determination of the (p, d, q) parameters, Auto_ARIMA automates this process. It employs an algorithmic approach to iteratively test different combinations of parameters and select the optimal set based on AIC.

2. **Model Building**

In building my Auto_ARIMA model, I employed two distinct methods to train and optimize the model's hyperparameters. My initial approach involved manually setting the range for the hyperparameters, instructing the algorithm to seek out the best possible combination. For the US dataset, this method was implemented using the command auto_arima(train_us, start_p=1, start_q=1, max_p=7, max_q=7, start_P=0, seasonal=True, d=1, trace=True, error_action='ignore', suppress_warnings=True, stepwise=True). This process yielded the parameter combination of (6,1,7).

However, I also experimented with a different approach, where I allowed the Auto_ARIMA model to autonomously determine the hyperparameters without any initial input from me. By simply using auto_arima(train_us, seasonal=True, stepwise=True), the algorithm provided a seemingly more effective combination for the US dataset: (2,1,5). This result prompted me to apply this second method to each country's data, ultimately tuning a model for each and calculating the RMSE for the entire dataset.

One of the main challenges I faced was the hyperparameter tuning. Initially, the manual input method seemed more controlled, but it potentially limited the model's ability to explore a wider range of parameter combinations. The resolution came with the adoption of Auto_ARIMA's default setting, which allowed for a more comprehensive search of parameter space, leading to potentially more accurate models.

3. **Model Performance**

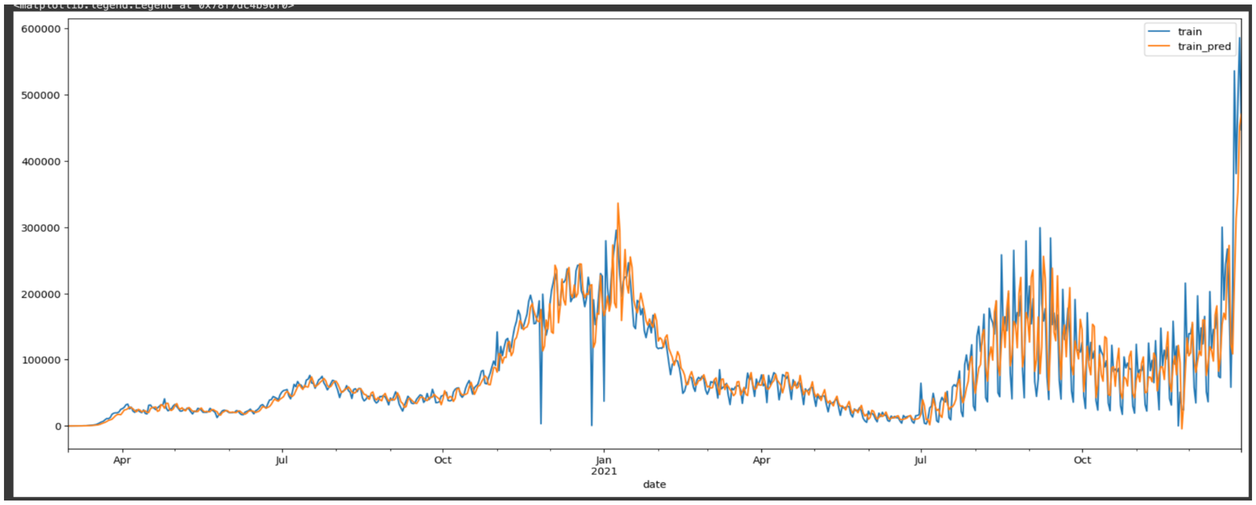
<u>Model focusing on the US:</u>

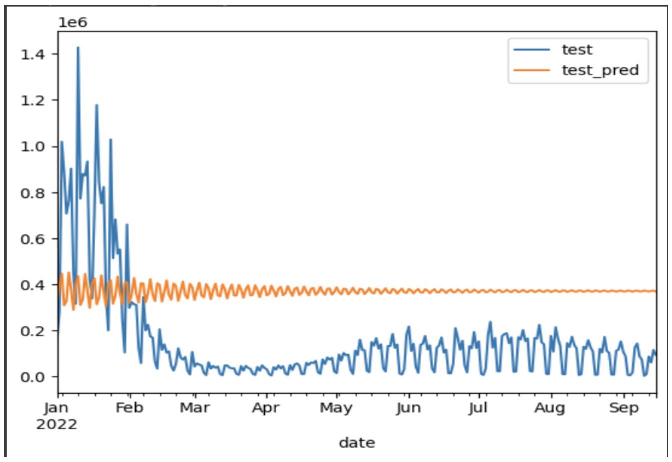
Train RMSE = 34609.33, MAPE = 489.13

Test RMSE = 308472.82, MAPE = 954.16

<u>The entire dataset:</u>

Train RMSE = 11039.53, Test RMSE = 135464.73





For the model focusing on the US, the test RMSE has reduced. This is because the Auto Arima model better handles the issue of overfitting, given its ability to automatically select parameters can lead to better generalization. Interestingly, when applying to the entire dataset, the arbitrary method of applying ARIMA outperformed tuning the Auto ARIMA model for each country slightly. The entire dataset, encompassing multiple countries, is likely more complex and varied than the data for a single country like the US. Auto_ARIMA's automated tuning, while effective for individual countries, might not capture the intricacies and interactions present in a more diverse and comprehensive dataset. More exploration should be done in the future to find the cause of this observation.

Prophet (Univariate) Model

1. Data Preparation

Prophet is specifically designed for forecasting time series data that exhibits strong seasonal patterns and various irregularities. One of its primary benefits is its robust handling of missing data and its ability to model holiday effects, which are not inherently accommodated by ARIMA models. The architecture of Prophet is based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It allows for a more flexible model fitting compared to the rigid structure of ARIMA models. Prophet is also a user-friendly model, meaning that it requires minimum manual tuning.

In preparing the dataset for the Prophet model, I followed similar preprocessing steps as those used for ARIMA and Auto_ARIMA models. This included removing data from the early months of the pandemic, acknowledging that the global outbreak of COVID-19 was not fully underway before this time, and grouping the data by country. However, an additional crucial step in the preparation was formatting the data to meet Prophet's requirements. Prophet necessitates a specific DataFrame format, with columns labeled 'ds' for the date/time component and 'y' for the metric being forecasted. This transformation was vital to enable the Prophet model to accurately parse the data and apply its complex algorithms for time series forecasting.

2. Model Building

In the construction of the Prophet (univariate) model, I adopted a streamlined training process, capitalizing on Prophet's inherent efficiency and accuracy in handling time series data. Prophet's design is such that it often performs exceptionally well with its default settings, reducing the need for extensive fine-tuning of parameters. This characteristic of the Prophet model significantly simplifies the model-building process.

For the training process, I initially focused on the United States as a representative case. I directly fitted the Prophet model to the U.S. dataset without additional parameter adjustments. This approach allowed me to evaluate the model's performance quickly and effectively. The visual analysis of the model's forecasts, along with the evaluation of performance scores, provided me with the necessary confidence in the model's accuracy and reliability. Satisfied with these initial results, I then proceeded to fit a separate Prophet univariate model for each country in the dataset.

One of the key strengths of the Prophet model is its robust default parameter settings, which are generally well-suited for a wide range of time series forecasting tasks. This aspect minimized the challenges typically faced in the model tuning phase. However, I remained vigilant for any signs of overfitting or underfitting, ready to make adjustments if needed. Fortunately, Prophet's performance with its default settings proved to be highly effective for my dataset, making the training process straightforward and efficient.

3. Model Performance

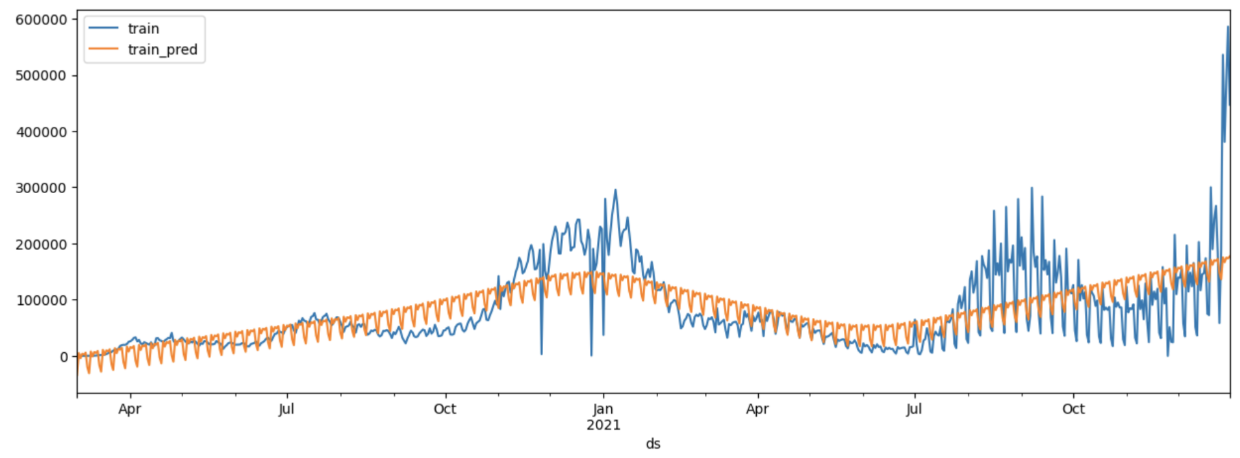
Model focusing on the US:

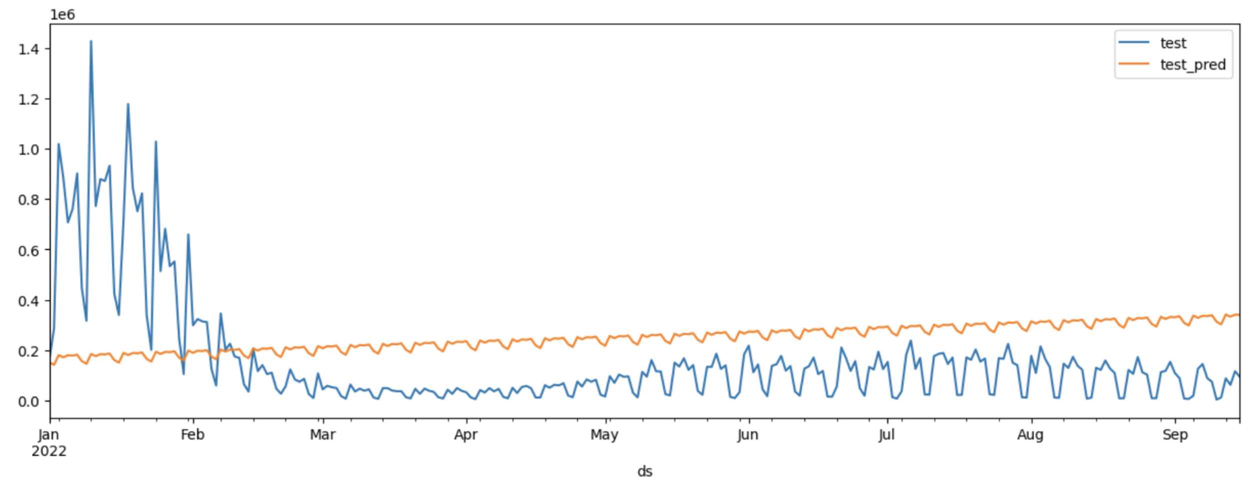
Train RMSE = 53767.83, MAPE = 1018.65

Test RMSE = 261758.29, MAPE = 609.12

The entire dataset:

Train RMSE = 17162.58, Test RMSE = 83536.54





For both models focusing on the US and the model for the entire dataset, Prophet(univariate) outperformed ARIMA models. It has also reduced the issue of overfitting.

Prophet (Multivariate) Model

1. **Data Preparation**

In addition to the same fundamental preprocessing steps as in previous models, a key enhancement in this phase was the addition of extra predictors to the model. An important addition was the inclusion of holiday data, which can significantly impact time series trends, especially in the context of COVID-19. I utilized Prophet's make_holidays function to integrate holidays into the dataset for most countries except for Qatar, which lacks a built-in holiday function in Prophet. I coded holidays with '1' on relevant dates.

Given my model's focus on the country level, it was necessary to modify certain features for relevance and accuracy. In the original dataset, features that were constant within each country, such as geographical or demographic data, were dropped. I also manually selected features based on common sense. For instance, when grouping data by countries, I summed features like population and area while averaging columns like life expectancy, ensuring these metrics accurately reflected country-level characteristics. Additionally, to handle missing values in lagging features, I inputted zeros.

2. Model Building

Recognizing Prophet's adeptness at handling complex time series data with its default settings, I chose not to engage in extensive manual tuning. This decision was based on the observation that the default tuning often yields satisfactory results, striking a balance between model complexity and forecast accuracy. Satisfied with the results from the initial trial with the US dataset, I proceeded to apply the Prophet (multivariate) model to each of the other countries in my dataset. To optimize performance, I carefully evaluated the model outputs, particularly focusing on how well the model captured the underlying trends and seasonalities in the data. Adjustments, if necessary, would have involved fine-tuning parameters related to trend flexibility and seasonality strength. However, in my case, the default settings proved to be highly effective.

In summary, the training process of the Prophet (multivariate) model was characterized by an emphasis on leveraging the model's default capabilities, with careful monitoring and evaluation to ensure optimal performance across various country datasets.

3. Model Performance

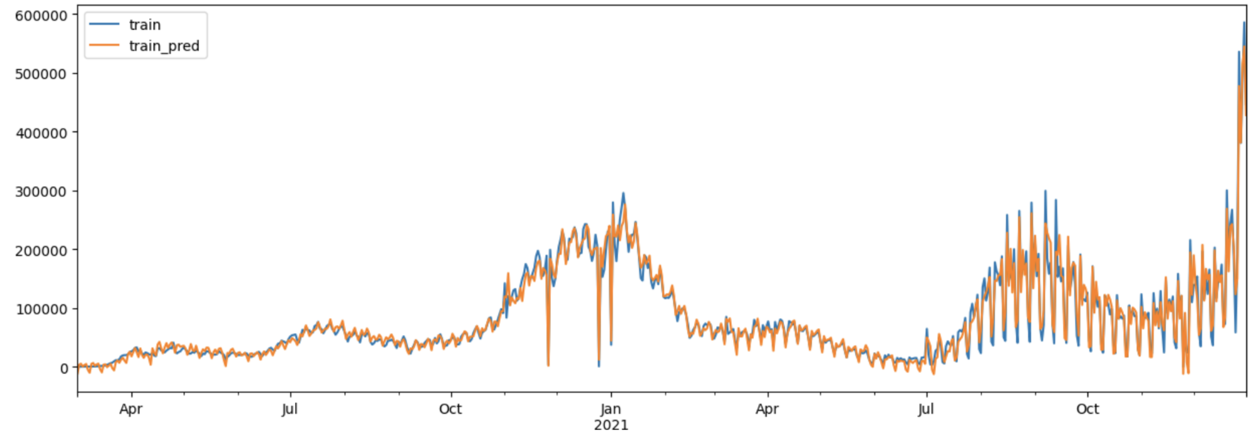
Model focusing on the US:

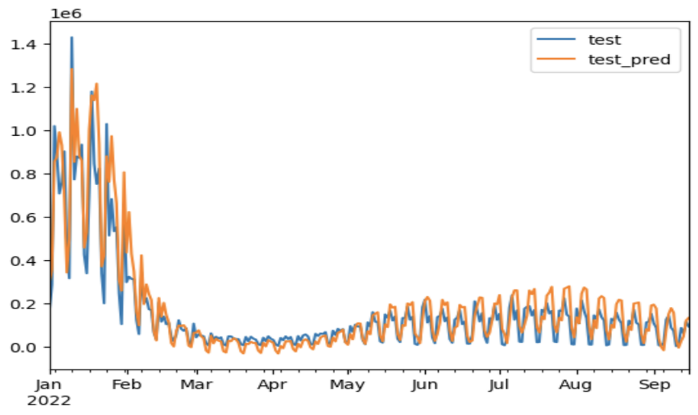
Train RMSE = 13978.21, MAPE = 223.68

Test RMSE = 82497.56, MAPE = 94.23

The entire dataset:

Train RMSE = 4781.66, Test RMSE = 26254.15





The model performance is excellent compared to the Prophet(univariate) model, both RMSE and MAPE for the US model and the final model reducing significantly. The superior performance of the Prophet (multivariate) model over its univariate counterpart can primarily be attributed to its ability to incorporate multiple predictors, offering a more holistic view of the data. This inclusion allows for a richer understanding of complex interactions and trends within the COVID-19 dataset. This approach reduces the risk of overfitting to single data trends and enhances the model's adaptability to diverse data sources and changing patterns, as reflected in the significant reduction in both RMSE and MAPE for the US model and the overall final model.

XGBoost Model

1. **Data Preparation**

Recognizing the unique handling of missing data by XGBoost, I replaced null values in critical columns like 'new_persons_fully_vaccinated' with large numbers for instances where vaccination data was not available. This strategy was chosen instead of typical imputation methods to maintain data integrity and reflect the absence of vaccination in certain periods or locations.

A significant shift in the modeling strategy was the transition from a country-level to a dataset-wide model. Unlike previous models which focused on individual countries, the XGBoost model was designed to analyze the entire dataset. To accommodate this, I employed One-Hot Encoding for the 'country_code' variable.

The predictors I used for XGboost model include 'date', 'country_code', 'new_deceased', 'cumulative_deceased', 'population', 'population_male', 'population_female', 'area_sq_km', 'life_expectancy', 'mobility_workplaces', 'new_persons_fully_vaccinated', 'Cumulative_persons_fully_vaccinated', 'new_confirmed_lag1', 'new_confirmed_lag3', 'new_confirmed_lag7', 'new_confirmed_mean3', 'new_confirmed_mean7', 'new_confirmed_std3', 'new_confirmed_std7', 'day_of_week', 'quarter', 'month', 'year', 'dayofyear', 'dayofmonth', 'weekofyear', 'season'. Also the information on the economic aspects in the original dataset is included.

2. Model Building

XGBoost operates as a powerful ensemble learning algorithm that utilizes gradient boosting frameworks, making it highly suitable for complex predictive modeling tasks like forecasting COVID-19 trends. Its key parameters, crucial for model optimization, include the number of trees (n_estimators), depth of each tree (max_depth), learning rate, regularization terms (reg_lambda and gamma), and subsampling rate (subsample).

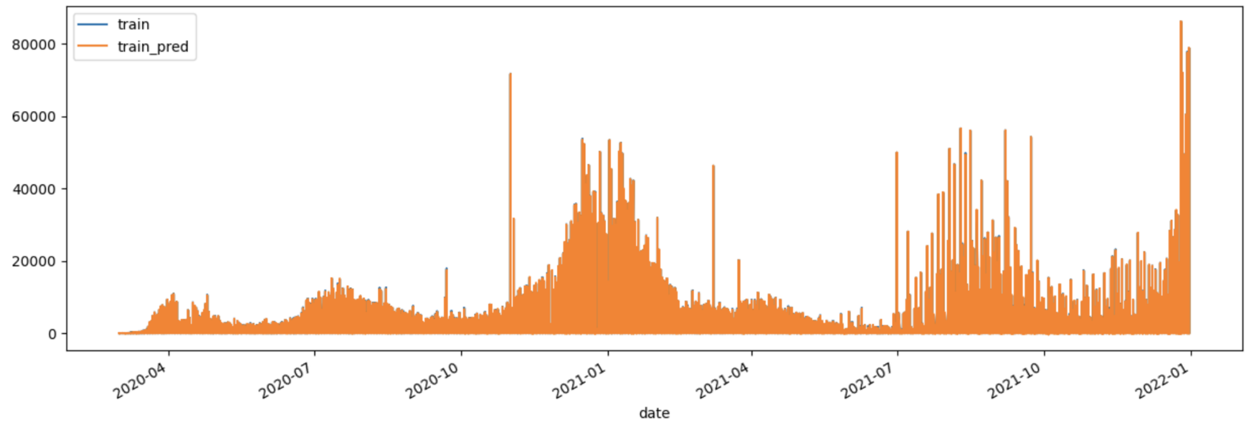
For the training process for the XGBoost model, initially, I visualized the relationship between various parameters—number of trees, tree depth, learning rate, and regularization terms—and the cross-validation error to determine appropriate ranges for each. This exploratory analysis informed the setup of my grid search, where I defined param_grid with ranges for 'max_depth', 'learning_rate', 'reg_lambda', 'n_estimators', 'gamma', and 'subsample'. After three rounds of fine-tuning, I selected the optimal parameter values: max_depth of 5, learning rate of 0.2, reg_lambda of 2, n_estimators at 4000, gamma at 1000, and a subsample rate of 1.

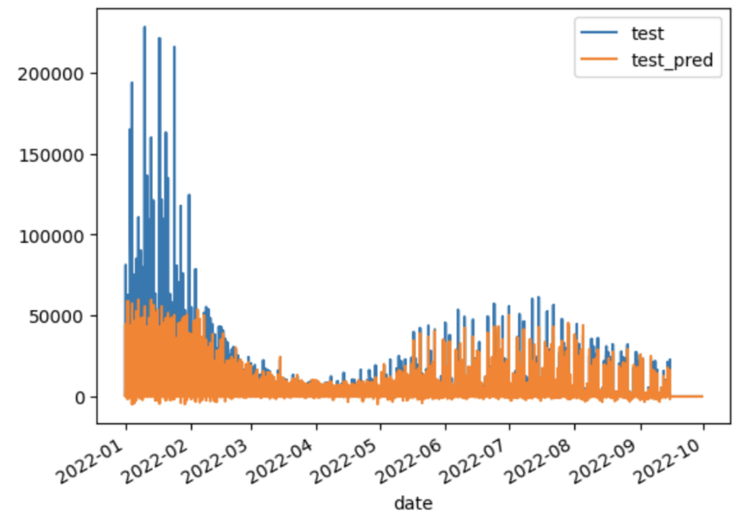
One of the significant challenges I faced during the training phase was managing the trade-off between the time required for hyperparameter tuning and the accuracy of the model. Extensive tuning can lead to better model performance but at the cost of increased computational time and resources. The visuals helped me a lot in minimizing the ranges for hyperparameter tuning.

3. Model Performance

Train RMSE = 58.47, Test RMSE = 3865.71

XGboost outperforms the previous models significantly for both train and test RMSE.





LSTM Model

1. **Data Preparation**

To begin with, the preparation of data for the LSTM model followed a approach similar to that used for the XGBoost model. The visualization of the target variable's distribution revealed a pronounced right skew in the data. To address this, a normalization process was implemented, a crucial step especially beneficial for neural networks like LSTM, known for their sensitivity to scale and distribution of input data. In addition to normalization, special attention was given to handling missing data. For lagging and window features, a method of imputation using zeros was employed.

To tailor the dataset for compatibility with the LSTM model, transformations were undertaken. The data was prepared in a format conducive to LSTM processing. This involved using the make_series function, shaping the data into a suitable structure. This function played a pivotal role in aligning the dataset with the unique requirements of LSTM models.

The selection of predictors for the LSTM model was aligned with those used in the XGBoost model. This consistency in predictor variables also allowed for a coherent and comparative analysis between the two models. The chosen predictors, having demonstrated their relevance and effectiveness in the XGBoost framework, were deemed equally valuable for the LSTM model.

2. **Model Building**

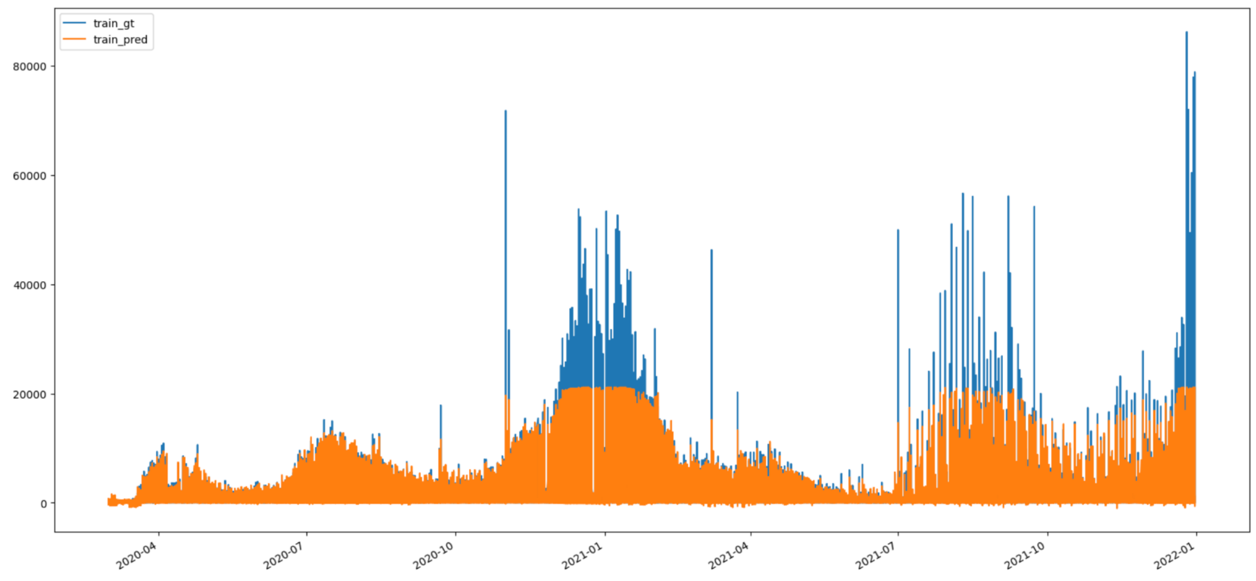
In building the LSTM model, I focused on tuning several key parameters. The key parameters include the learning rate, which determines the step size at each iteration while moving toward a minimum of the loss function; batch size, dictating the number of training examples used in one iteration; the number of epochs, representing the number of complete passes through the training dataset; and sequence length, which is the length of the input sequences for the LSTM.

The training process involved utilizing RandomizedSearchCV for hyperparameter tuning, conducted two rounds to identify the optimal settings. The final parameters chosen were a learning rate of 0.01, a batch size of 32, 100 epochs, and a sequence length of 14.

Throughout the training, I encountered and overcame two major challenges. Initially, the model fitting process resulted in a loss value of 'nan', which I traced back to the presence of missing values in the dataset. By imputing zeros for all missing values, I resolved this issue, ensuring that the model could process the data effectively. The second challenge arose during the prediction phase on the test data, where I again encountered 'nan' predictions. This issue was addressed by ensuring that the preprocessing steps applied to the training data were precisely replicated for the testing data. I learned the importance of consistency in data preprocessing—any discrepancies, particularly in normalization, scaling, and handling of missing values, can lead to unpredictable and erroneous model outcomes.

3. Model Performance

Train RMSE = 1040.43, Test RMSE = 4102.34



Results

	US train RMSE	**US test RMSE**	**Train RSME**	**Test RMSE**
ARIMA	**28633.74**	**382118.47**	**9134.10**	**123881.36**
Auto ARIMA	**34609.33**	**308472.82**	**11039.53**	**135464.73**
Prophet(Univariate)	**53767.83**	**261758.29**	**17162.58**	**83536.54**
Prophet(Multivariate)	**13978.21**	**82497.56**	**4781.66**	**26254.15**

XGBoost	**n/a**	**n/a**	**58.47**	**3865.71**
LSTM	**n/a**	**n/a**	**1040.43**	**4102.34**

Based on the results, XGBoost is the best performing model, followed by LSTM, Prophet(multivariate), Prophet(Univariate0, and ARIMA models.

XGBoost: As the best-performing model in this analysis, XGBoost's strength lies in its ability to handle a variety of data types, deal with missing values effectively, and capture complex nonlinear relationships between features.However, XGBoost can be computationally intensive and may require extensive hyperparameters to avoid overfitting in certain scenarios.

LSTM: The LSTM model, ranking second in performance, excels in capturing temporal dependencies and patterns over time, making it particularly suitable for time series forecasting like COVID-19 case predictions.The main drawback of LSTM is its complexity and the need for extensive training data. It can also be challenging to tune and requires significant computational resources.

Prophet (Multivariate): The multivariate version of Prophet offers the advantage of incorporating multiple predictors, enhancing its ability to model complex relationships and interactions in the data. It is user-friendly and handles seasonality and trends effectively. However, its performance may be limited by its additive model structure, which might not capture more complex interactions as effectively as models like XGBoost.

Prophet (Univariate): The univariate Prophet model is straightforward to implement and excels in capturing seasonality and trends with less data preprocessing. Its intuitive nature and ease of use are its main strengths. However, it lacks the ability to incorporate multiple predictors, which can limit its predictive power compared to multivariate approaches.

ARIMA: The ARIMA model excels in analyzing time series data with clear trends or seasonal patterns due to its simplicity and interpretability. Ideal for stationary series, its main limitation lies in handling complex, nonlinear datasets and its inability to incorporate external variables or manage missing data effectively.

Auto_ARIMA: Building upon ARIMA, Auto_ARIMA automates parameter selection, significantly simplifying the modeling process. While it offers greater user-friendliness and efficiency in model fitting, it shares ARIMA's limitations.

Discussion

In my analysis, one noticeable outcome was the slightly superior performance of the ARIMA model, where hyperparameters were arbitrarily set for each country, compared to the Auto_ARIMA model tuned individually for each country. This suggests that while automated tuning is efficient, it may not always capture the nuances of specific datasets as effectively as manual tuning. Moreover, finer tuning of models is definitely required for some of my models to achieve better performance and reduce overfitting. Due to the time limit, I didn’t get to tune every model very carefully. Understanding and implementing various models, particularly the

LSTM, presented a significant learning curve. Grasping the concepts and intricacies of these models required dedicated time and effort, which was a challenging yet enriching part of the process.

I absolutely gained valuable experience working on this project. Working with models like XGBoost, LSTM, Prophet, and ARIMA, most of which were new to me, provided a valuable learning experience. I gained knowledge in machine learning and time series analysis by leveraging online resources and engaging in discussions with my group members. This project enhanced my ability to quickly assimilate new information and apply it effectively in a practical setting. I particularly appreciated the collaborative nature of this group project. The process of mutual learning and sharing perspectives was valuable in achieving our analytical goals.

Conclusion

Among the various models tested, XGBoost emerged as the most effective, demonstrating its robustness in handling complex, structured data.

Personally, I developed a preference for the Prophet (Multivariate) model throughout the course of this project. While its RMSE did not reach the lows of the XGBoost and LSTM models, its overall efficiency in terms of resource utilization, time consumption, and manual effort involved stood out as particularly advantageous. The Prophet (Multivariate) model requires significantly less computational resources and manual tuning compared to the other models, making it an attractive option in a professional setting.

The project underscored the value of collaborative work in data science. Working in a group allowed for diverse approaches to be applied to the same dataset, facilitating a richer analysis and a more comprehensive understanding of the models and the data. My group members helped me identify my mistakes in building the model and helped me debug. Specifically, I learned from Danling how to build a Dataframe and store each country’s prediction and then calculate the RMSE on the entire dataset.

References

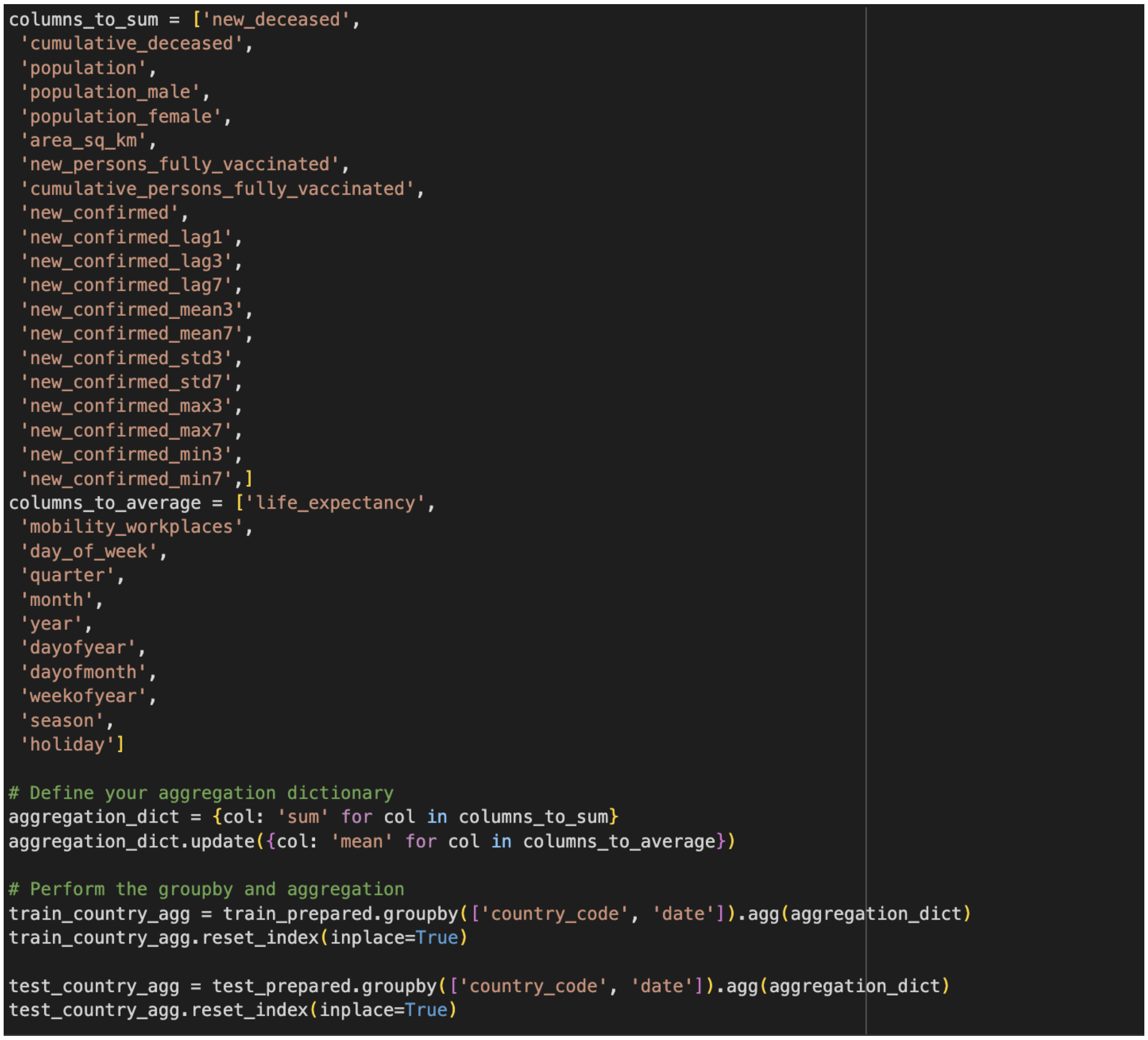
Long short-term memory (LSTM) RNN in Tensorflow. (2023, January 8). GeeksforGeeks. https://www.geeksforgeeks.org/long-short-term-memory-lstm-rnn-in-tensorflow/

Python LSTM (Long Short-Term Memory Network) for Stock Predictions. (n.d.). Www.datacamp.com. https://www.datacamp.com/tutorial/lstm-python-stock-market

Appendices

code snippets

Predictors chosen for multivariate prophet:

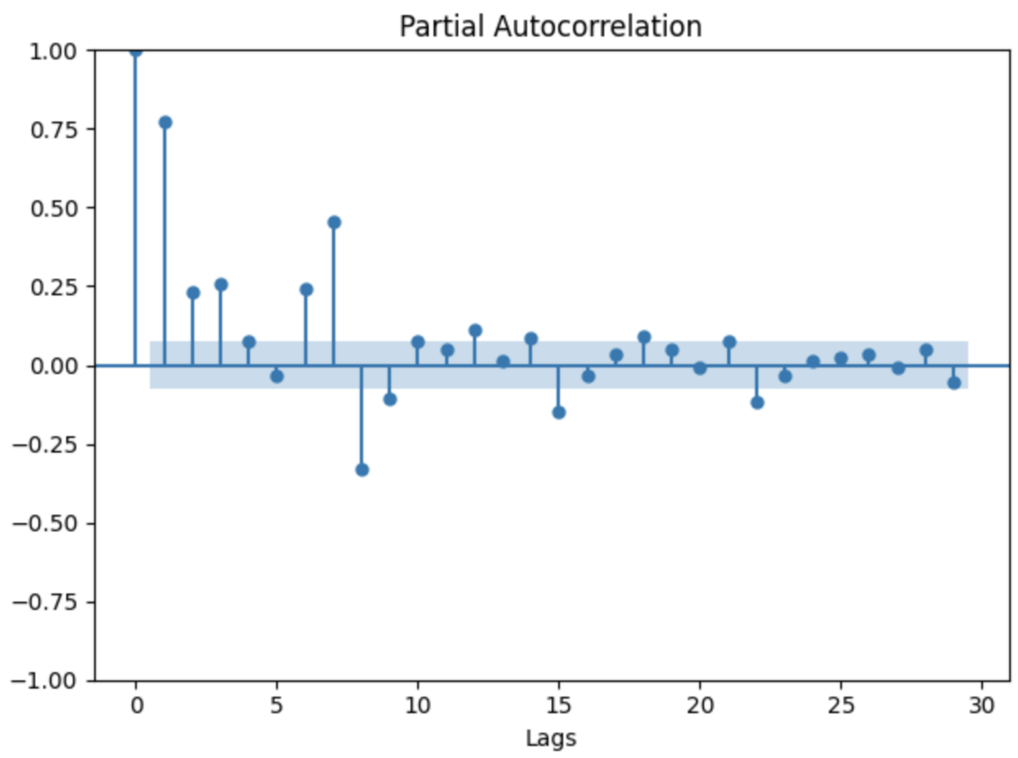


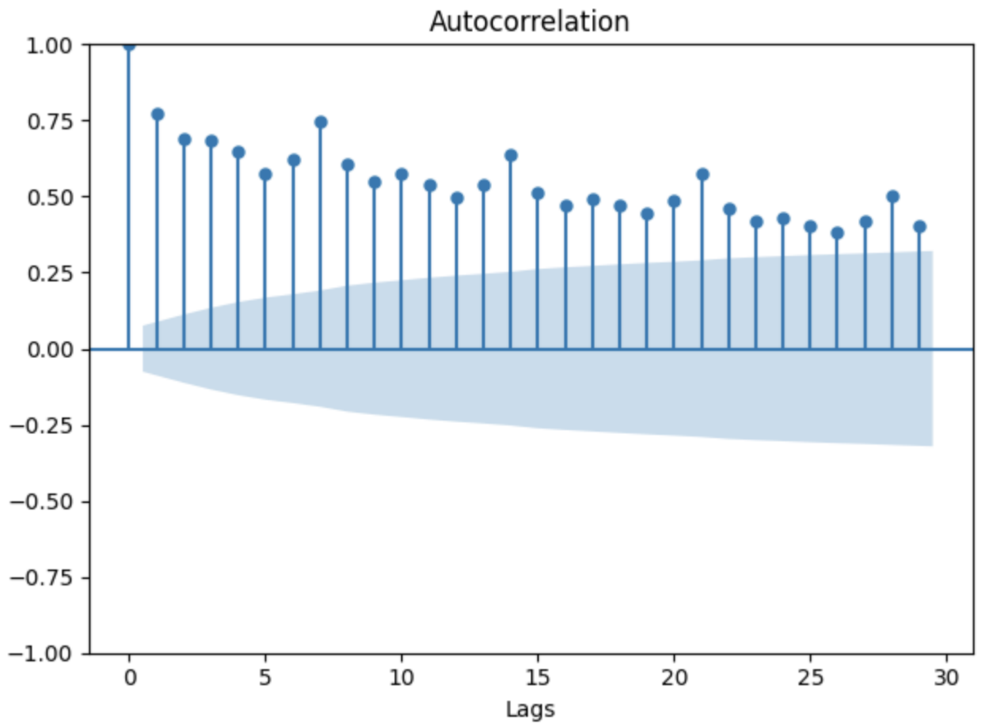
```
columns_to_sum = ['new_deceased',
 'cumulative_deceased',
 'population',
 'population_male',
 'population_female',
 'area_sq_km',
 'new_persons_fully_vaccinated',
 'cumulative_persons_fully_vaccinated',
 'new_confirmed',
 'new_confirmed_lag1',
 'new_confirmed_lag3',
 'new_confirmed_lag7',
 'new_confirmed_mean3',
 'new_confirmed_mean7',
 'new_confirmed_std3',
 'new_confirmed_std7',
 'new_confirmed_max3',
 'new_confirmed_max7',
 'new_confirmed_min3',
 'new_confirmed_min7',]
columns_to_average = ['life_expectancy',
 'mobility_workplaces',
 'day_of_week',
 'quarter',
 'month',
 'year',
 'dayofyear',
 'dayofmonth',
 'weekofyear',
 'season',
 'holiday']

# Define your aggregation dictionary
aggregation_dict = {col: 'sum' for col in columns_to_sum}
aggregation_dict.update({col: 'mean' for col in columns_to_average})

# Perform the groupby and aggregation
train_country_agg = train_prepared.groupby(['country_code', 'date']).agg(aggregation_dict)
train_country_agg.reset_index(inplace=True)

test_country_agg = test_prepared.groupby(['country_code', 'date']).agg(aggregation_dict)
test_country_agg.reset_index(inplace=True)
```


Partial Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
30
Lags


Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
30
Lags