VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
MASTER'S STUDY PROGRAMME
MODELLING AND DATA ANALYSIS

Evaluation of Consumer Confidence Indicators Using Social Media and Administrative Data

Master's thesis

Author: Akvilė Vitkauskaitė
VU email address: akvile.vitkauskaite@mif.stud.vu.lt
Supervisor: Andrius Čiginas

Vilnius

2024

Abstract

The main objective of this study is to nowcast and forecast the Consumer Confidence Index (CCI). The aim is to estimate the current month's CCI values faster than those obtained using the traditional survey methodology, which usually provides results at the end of the month. For instance, while the official CCI for November would typically be available in the last few days of November, this research aims to provide an early estimate at the beginning of November, utilizing data collected at the start of the month. This is achieved by combining key economic indicators with historical CCI values. The research includes examining the relationship between traditional survey-based indicators and consumer sentiment expressed on social media platforms. Social media expressions, particularly from X (Twitter), are analyzed through its official API. The sentiment analysis of tweets has enabled us to create a Social Media Indicator (SMI) that offers a distinct advantage in our predictive models. In addition, the study explores the possibility of integrating key economic indicators from administrative data, such as inflation rate, income statistics, and unemployment. In general, obtaining data for research from popular social platforms such as Facebook and Instagram is not possible due to stringent privacy policies and data protection regulations. Nevertheless, data are easily and legally available from X, but this platform is not so popular in Lithuania. Therefore, the representativeness of X data raises special issues. Taking everything into account, by combining traditional economic indicators with advanced sentiment analysis from X, the study seeks to deliver prompt CCI predictions ahead of standard survey timelines.

Keywords: Consumer confidence, Social Media, Twitter, Sentiment Analysis, SARIMAX, VECM, Random Forest, XGBoost

Santrauka

Šio tyrimo pagrindinis tikslas — prognozuoti vartotojų pasitikėjimo rodiklį (VPR). Siekiama įvertinti einamojo mėnesio VPR reikšmes greičiau nei tai daroma naudojant tradicinius apklausų metodus, kurie paprastai rezultatus pateikia mėnesio pabaigoje. Pavyzdžiui, nors oficialus lapkričio mėnesio VPR paprastai būna skelbiamas lapkričio mėnesio paskutinėmis dienomis, šis tyrimas siekia pateikti ankstyvą įvertį jau lapkričio pradžioje, naudojant duomenis, surinktus mėnesio pirmomis dienomis. Tai pasiekiama derinant pagrindinius ekonominius rodiklius su istorinėmis VPR reikšmėmis. Tyrimas apima ir sąryšių tarp tradicinių, apklausomis grindžiamų rodiklių bei vartotojų nuomonės, išreikštos socialiniuose tinkluose, analizę. Vartotojų sentimentai yra gaunami iš socialinio tinklo X (Twitter) naudojant oficialių API. Tviterio žinučių sentimentų analizė leidžia mums sukurti socialinės medijos indikatorių (SMI), kuris yra naudingas prognozavimo modeliams, nes padidina prognozės tikslumą, atspindėdamas socialiniuose tinkluose vyraujančias nuotaikas. Šiame darbe iš administracinių duomenų bandome integruoti pagrindinius ekonominius rodiklius: infliacijos lygį, pajamų statistiką ir nedarbo apimtis. Verta paminėti, kad gauti duomenis iš populiarių socialinių platformų, tokių kaip Facebook ir Instagram, neįmanoma dėl griežtos privatumo politikos ir duomenų apsaugos reglamentų. Tuo tarpu, duomenys iš X yra lengvai ir legaliai prieinami, tačiau Lietuvoje ši platforma nėra tokia populiari. Todėl kyla tam tikrų X duomenų reprezentatyvumo problemų. Taigi, šis tyrimas siekia pateikti VPR prognozes anksčiau nei rezultatai gaunami tradiciniais metodais, derinant tradicinius ekonominius rodiklius ir pažangią sentimentų analizę X duomenims.

Raktiniai žodžiai: Vartotojų pasitikėjimo rodiklis, Socialiniai tinklai, Twitter, Sentimentų analizė, SARIMAX, VECM, Random Forest, XGBoost

1 Introduction

Consumer confidence is a vital economic indicator that influences the decision-making processes of policymakers, businesses, and investors, providing valuable insights into individuals' sentiments and expectations regarding the state of the economy [21]. Traditionally, consumer confidence indicators have been derived from survey data, capturing the opinions and perceptions of individuals through structured questionnaires [31, 12, 36].

With the rise of social media platforms and the abundance of user-generated content, there is a new opportunity to assess consumer sentiment in new ways. Platforms like internet blogs, discussion forums, and social media sites enable individuals to openly express their opinions and experiences [1]. By analyzing this rich data through sentiment analysis techniques, valuable insights can be gained regarding public opinion and attitudes, including the evaluation of consumer confidence. These insights can inform decision-making in various fields such as market research and business strategies.

In addition to social media data, administrative data from government agencies, such as employment statistics, income records, and other relevant economic indicators, can provide a rich and objective source of information. By incorporating these auxiliary variables, it can potentially improve the accuracy and timeliness of consumer confidence indicators. The study by Curtin (2007) [11] emphasizes the significant relationship between changes in the unemployment rate and consumer sentiment, highlighting the powerful influence of employment conditions on consumer expectations.

This master thesis aims to evaluate the effectiveness and reliability of consumer confidence indicators derived using social media data and administrative data. Also, an examination of the relationships between consumer sentiment expressed on social media platforms and traditional survey-based indicators is performed. Furthermore, the potential of using administrative data as auxiliary variables to enhance the forecasting accuracy of consumer confidence indicators is explored.

We aim to bridge the gap between traditional survey-based indicators and emerging data sources by evaluating and utilizing social media and administrative data to measure and forecast consumer confidence. The objective is to provide valuable insights and contribute to the advancement of economic forecasting and decision-making in an increasingly digitalized world.

Our study used SARIMAX, VECM, Random Forest, and XGBoost to predict the CCI from social media and administrative data. XGBoost yielded the highest accuracy, SARIMAX followed closely, and Random Forest and VECM were comparatively less precise. We are going to present a detailed analysis and comparison of these models.

The thesis is divided into several sections. Section 2 provides a literature review that examines the historical context and evolution of consumer confidence indicators, the emerging role of social media data, and the inclusion of administrative data as auxiliary sources. Section 3 introduces the data used in this study, detailing the sources, collection methods, and pre-processing steps. Section 4 conducts an exploratory analysis. Section 5 describes the evaluation metrics and rolling forecasting method, setting the stage for Section 6, which discusses the development and implementation of various forecasting models. Finally, Section 7 summarizes the findings, compares the models' performance, and suggests directions for future research. Appendices include supplementary figures, tables, and Python code.

2 Literature review

2.1 Historical development of consumer confidence indicators

In the middle of the twentieth century, consumer confidence measurements became highly important for predicting economic trends. They gained recognition among businesses and government economists as a reliable method to forecast market behaviour and understand consumer choices. This led to a greater emphasis on comprehending consumers, including their attitudes, expectations, and the psychological factors that shape their economic decisions [28].

The development of the CCI requires the collaboration of various organizations and institutions. George Katona and Rensis Likert pioneered the initial methods to gauge consumer confidence in the late 1940s, aiming to integrate tangible measurements of expectations into models analyzing spending and saving patterns [23]. Mueller (1963) [31] played a significant role in advancing consumer confidence indices by introducing the concept of using them to forecast consumption patterns. Mueller's innovative research involved analyzing data from the Michigan survey of consumers over a period of ten years. Her findings demonstrated the vital role of consumer confidence in explaining spending habits, as evidenced by its inclusion in a regression model that considered previous consumption levels [31].

The Consumer Confidence Board in the USA has played a crucial role in the development of consumer confidence indices, with one notable index being the CCI introduced in 1967. Their survey evaluates individuals' perspectives on current and future economic conditions, as well as their employment prospects. The CCI is widely recognized as an indicator of the strength and stability of the U.S. economy. In May 2021, the Conference Board Consumer Confidence Survey transitioned from being administered by The Nielsen Company to Toluna, a technology company with a large consumer panel of over 36 million individuals. Before November 2010, the survey was conducted by TNS through mail [36].

The contemporary administration of the CCI in the EU lies under the purview of Eurostat, the statistical office of the European Union, with monthly calculations conducted across all member countries [16]. Since May 2001, the State Data Agency (Statistics Lithuania) [26] has been responsible for conducting similar surveys in Lithuania. The main objective of the consumer opinion statistical survey is to obtain information regarding consumers' intentions to make purchases, their saving capabilities, as well as their perceptions of the economic situation and its influence on their intentions.

2.2 The role of social media data in estimating consumer confidence

The integration of social media data in the estimation of consumer confidence indicators has gained significant attention in recent years. Innovative and accelerated use of big data sources has become particularly important in addressing challenges, exemplified by the increased frequency and application of Twitter data in official statistics during the COVID-19 crisis [5]. Alternative data sources, including social media platforms such as Twitter and Facebook, have emerged as valuable resources for capturing real-time information reflecting public sentiment and opinions [18, 37]. In the realm of official statistics, notable efforts have been made to explore the use of social media data for constructing consumer confidence indices. Istat's Social Mood on Economy Index in Italy, developed by Catanese et al (2022)[8] incorporates social media data alongside traditional survey-based measures. Analyzing the sentiment expressed in tweets related to the economy, this index provides a complementary perspective on consumer confidence. Additionally, van den Brakel et al. (2017) [6] investigated the utilization of alternative data sources in the Dutch Consumer Confidence Survey. Their research developed a multivariate structural time series modelling methodology that incorporates social media data and repeated survey data. This approach aims to enhance the accuracy and timeliness of consumer confidence estimates, even in the absence of sample data.

While there is potential for using social media data in estimating consumer confidence in official statistics, challenges such as data noise, representativeness, and the development of reliable methodologies still need to be addressed [35]. Ongoing research efforts [4] continue to explore innovative approaches, proposing a novel method of social media analytics called NACOP. By utilizing big data and data science techniques, NACOP analyzes social media data on purchasing behaviour, jobs/employment, consumer price, and personal finance. This approach aims to provide more accurate insights into the true state of the economy, offering significant implications for researchers and practitioners.

2.3 Administrative data as auxiliary data for estimating CCI

Nowzohour and Stracca (2017) [33] find that the CCI is highly correlated with expectations on the general economic situation, unemployment, and the financial situation. Consumer confidence is positively associated with future inflation, industrial production growth, real house price growth, and real appreciation, while negatively associated with unemployment and short-term interest rates. Demirel and Artan (2017) [14] employ panel causality analysis to examine the causality relationships between economic confidence and fundamental macroeconomic indicators in selected European Union countries. A unidirectional causality is observed from real exchange rate and interest rate to economic confidence, and from economic confidence to unemployment. The results highlight the significant impact of confidence on production, consumption, inflation, and unemployment. The findings emphasize the importance of confidence as a psychological factor in economic decision-making.

The previously mentioned research provides insights and inspiration to explore the use of auxiliary data, such as unemployment data, as a means to enhance the accuracy of CCI forecasting models. The observed correlations between these auxiliary variables and the CCI suggest the potential benefits of incorporating them into the forecasting process.

3 Data and dataset

3.1 Overview

The dataset used in this study is designed to forecast the CCI faster than traditional approaches. CCI (Fig. 1) is assessed by a survey and it is calculated as the arithmetic mean of four balanced estimates: changes in household financial situations over the past and upcoming 12 months, anticipated changes in the national economy over the next 12 months, and variations in spending on significant purchases (like furniture or household appliances) compared to the previous year [26]. This dataset combines public sentiment from social media with key economic indicators. It contains monthly data from 2018 until November 2023, including the COVID-19 pandemic and geopolitical developments - Russia's invasion of Ukraine in February 2022. The observed declines in the CCI correspond to these periods of global and regional turbulence.

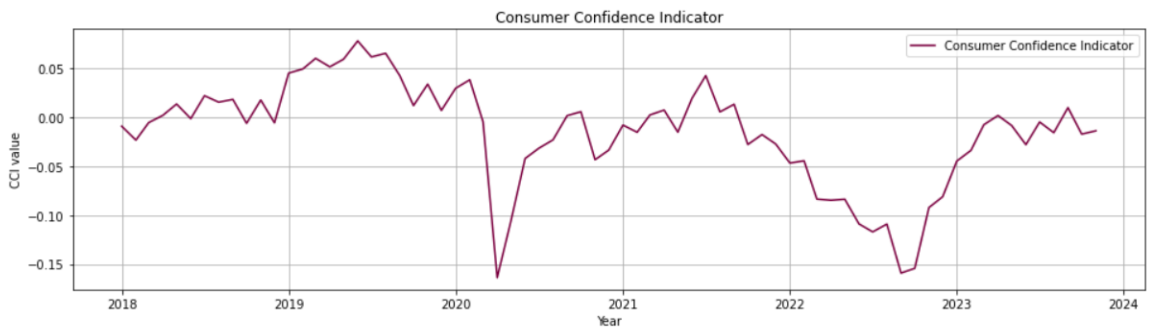


Figure 1: Consumer confidence indicator

3.2 Social media data (SMI)

We aims to refine the nowcasting of Lithuania's CCI by integrating social media data. Central to this integration is the Social Media Indicator (SMI), an index constructed from X data that quantifies public economic sentiment, offering real-time insights to complement traditional CCI measures. The social media component, derived from X using the platform's official API, captures the economic sentiment of Lithuanians through a crafted set of economic keywords (see Table 4).

As an auxiliary variable, the SMI enriches the dataset to increase the accuracy of the CCI forecasts.

3.2.1 Data gathering and preprocessing

In the middle of 2023, tweets reflecting Lithuania's economic sentiment began to be collected. The comprehensive list of 101 economic keywords and their variations was created from the words of official survey questionnaires and related research. Using this list, tweets were gathered weekly. To overcome the API's limitation of retrieving only the most recent 7-day-old tweets based on these keywords, the timelines method was used. This approach involves extracting the historical tweets directly from the timelines of users who had recently tweeted with our selected keywords, based on the assumption that their past tweets would likely contain similar economic content.

Before sentiment analysis is applied, the collected tweets go through several preprocessing steps. These steps are critical to ensure data quality and suitability for sentiment analysis. Initially, Twitter handlers, hashtags, URLs, special characters, and single characters are removed to clean the text. After that, multiple spaces are replaced with a single space, and all empty lines are replaced with NaNs (special floating point values in Python that are used to represent undefined or unrepresentable values). After dropping rows with missing values (NaNs), we excluded non-Lithuanian tweets (as Lithuanian and other languages have linguistic similarities, some non-Lithuanian tweets were collected).

3.2.2 Sentiment analysis

The final but not least step in developing the SMI is the sentiment analysis. Sentiment analysis, known as opinion mining, is a part of natural language processing (NLP). It identifies and categorises the opinions expressed in a text, particularly to determine whether the writer's view on a particular topic is positive, negative, or neutral. The purpose of sentiment analysis is to understand the attitudes, emotions and opinions from textual data [27].

Each tweet's text was analysed using five different techniques: TextBlob, Vader, Afinn, Flair, and Transformers. These analyzers evaluate the emotional sentiment of the text and assign scores that classify the sentiment as positive, neutral, or negative, except for Flair, which distinguishes only between positive and negative. Using multiple methods provides a broader and more nuanced perspective on public sentiment. Each different analyzer's sentiment index has slightly different relationships with the CCI.

The average sentiment score is calculated for Twitter users who tweet more than once a month. This method treats each tweet equally because each tweet has the same weight. This provides a balanced view of that user's economic sentiment throughout the month, no matter how often they tweet.

Each user's monthly sentiment scores are then categorized as either positive, neutral or negative (for Flair: positive or negative). The final SMI for the month is determined by calculating the balance of these sentiments. The number of negative expressions is subtracted from the total number of positive expressions. This figure is then divided by the sum of all positive and negative expressions, converting it into a percentage. Then SMI is adjusted to match the scale of the CCI, as the raw SMI values are typically ten times larger than the CCI values.

The interpretation of the Social Media Indicator (SMI) is that a positive sentiment score indicates an optimistic public mood regarding the economy, while 0 reflects a neutral mood, a negative - a pessimistic public sentiment.

This calculation is performed for each month in the dataset, spanning from January 2018 to November 2023 (71 months). As a result, five separate SMI indices emerge, each corresponding to one of the sentiment analysis techniques employed.

3.2.3 Sentiment analysis tools

Here we provide a brief overview of each sentiment analysis tool based on the general methodologies they employ.

- **TextBlob**. TextBlob simplifies text processing in Python, with sentiment analysis that evaluates text polarity and subjectivity. Polarity is a float between $[-1, 1]$, where -1 signifies negative and $+1$ positive sentiment. Subjectivity, ranging from $[0, 1]$, indicates how much personal opinion or emotion is expressed in the text. It is most suited for general text analysis including social media and product reviews [29].

- **Vader**. Vader (Valence Aware Dictionary for sEntiment Reasoning) is a rule-based sentiment analysis tool that is designed for social media sentiment analysis. It uses a combination of a sentiment lexicon and grammatical rules. It scores texts by evaluating slangs, emojis, and colloquial language, with a compound score indicating the overall sentiment. This compound score is normalized to range between $[-1, 1]$, where higher positive values denote more positive sentiment. It is effective for tweets and online reviews [19].

- **Transformers**. In this study, we utilize the Transformers library, developed by Hugging Face, which uses advanced models like BERT (Bidirectional Encoder Representations from Transformers) for deep contextual understanding. Specifically, we employ the "finiteautomata/bertweet-base-sentiment-analysis" model from the Transformers library for sentiment analysis. This particular model is a specialized adaptation of BERT, explicitly tailored for analyzing Twitter data. It effectively outputs a sentiment classification – positive, negative, or neutral – based on its contextual interpretation of the text, making it highly suited for understanding the unique linguistic characteristics of tweets [39].

- **Flair**. We utilize the Flair 'en-sentiment' model which is a pre-trained model for sentiment analysis provided by the Flair framework. This model excels in interpreting the sentiment of Twitter data. It provides a detailed sentiment score that reflects the intricacies of language use on social media platforms. Flair's approach to sentiment analysis is comprehensive as it gives a sentiment label along with a confidence score to signify the model's certainty. This method is particularly valuable when analyzing texts that require an understanding of deep contextual meanings, such as the brief and dynamic content of tweets [3].

- **Afinn**. Afinn employs a simple lexicon-based approach, where each word in the text is scored with a predefined sentiment value. These scores are then summed up to give an overall sentiment score for the text, with positive scores indicating positive sentiment and negative scores indicating negative sentiment [32]. As it relies on individual word scores without considering context means it does not capture the full context and may miss nuanced or ironic expressions. Despite this, Afinn is highly suitable for real-time sentiment analysis, such as monitoring Twitter during fast-moving events or trends. Its quick processing capability makes it efficient for analyzing straightforward texts and capturing the general sentiment direction, a useful feature for immediate public opinion assessment [10].

3.3 Administrative data

Alongside the SMI, the final dataset is enriched with a set of administrative economic indicators. All data for new indicators development is taken from the State Data Agency's official Database of Indicators while the number of unemployed is taken from the Lithuanian Public Employment Service. These variables are key in providing traditional economic signals. Below are listed indicators of the final dataset with explanations of how each was developed.

- **Average Wage**. It is made from a quarterly indicator, named "Average earnings", which is further disaggregated using linear interpolation into months and expressed in euros. It reflects the gross monthly earnings across the economy, providing insight into the earning power and potential spending capacity of the Lithuanian workforce.

- **Pension Data**. It is developed from the "Average state social insurance old-age pension" which is also a quarterly indicator. Given that significant changes in pension averages tend to occur at the start of the year, the average of the whole quarter was assigned to each month of the quarter possibly with insignificant information loss while preserving the dataset's integrity and having monthly data.

- **Inequality Indicator**. Calculated as the ratio of the average pension to the average wage, this indicator serves as a simple measure of income inequality. It is particularly useful as it potentially stands independent of price level fluctuations.

- **Inflation Rate**. It is the "Consumer Price Index (CPI) - based average annual inflation" indicator when set against the corresponding period of the previous year. This monthly data does not require any additional modifications. It is an extremely important measure that reflects the changing cost of living and its impact on consumer purchasing power.

- **Unemployment without seasonality**. The first unemployment metric included is the "Seasonally adjusted unemployment rate." This figure is reported monthly and reflects the unemployment rate after adjustments for seasonal labour market fluctuations.

- **Unemployment rate**. The second measure of unemployment is calculated by dividing the "number of unemployed at the end of the period" by the "Resident population at the beginning of the month." The numerator is provided by the Lithuanian Public Employment Service, offering a monthly count of individuals actively seeking employment. The denominator, sourced from the State Data Agency, gives the population size at the start of each month, providing a standard base for calculating the unemployment ratio.

These selected variables are combined into the final dataset to provide a comprehensive picture of the economic conditions that influence consumer confidence. The dataset, thus, not only reflects the sentiment captured through social media but is also grounded in the concrete economic realities as represented by these indicators.

3.4 Limitations of data

Twitter's user base in Lithuania, as reported in early 2023, stood at approximately 389,000, accounting for 14.2% of the overall population. This statistic becomes slightly more significant when considering Twitter's user age restriction. With the platform limiting its use to individuals 13 years and older, the 389,000 figure actually represents about 16.4% of the "eligible" audience in Lithuania. These insights, drawn from the "DIGITAL 2023: LITHUANIA" report by Simon Kemp [24], primarily reflect Twitter's advertising reach, which may not fully align with the actual active user base.

Additionally, the inclusion of “non-human” accounts in these figures complicates the true representation of user demographics. The fluctuating nature of Twitter's user metrics, such as the notable increase in 2022 followed by a decrease in early 2023, also necessitates a careful interpretation of these statistics, especially when utilized for economic sentiment analysis.

On the other hand, the integrity and representativeness of the Twitter data, which is critical to this study, was based on a cointegration test with the CCI. Cointegration refers to a statistical relationship where two or more time series, although individually non-stationary (their statistical properties like mean and variance change over time), move together in a long-term, stable manner. The test revealed a meaningful cointegration, confirming that Twitter data is important in assessing user sentiment in Lithuania.

The administrative data, while providing a solid economic backdrop, is not without its limitations. Certain variables like the population and the seasonally adjusted unemployment rate are initially reported as provisional for the last few months and may be subject to later revisions. Additionally, regarding the seasonally adjusted unemployment rate, only the data from the previous month is published at the end of each month. Therefore, for the November data, the same value as that of October is utilized as of the data collection cut-off in the middle of December 2023.

Such limitations, inherent in both social media and administrative data sources, could affect the precision of the nowcasting results. As data sources become more refined and comprehensive (for example, the State Data Agency will provide monthly data along with quarterly data), the potential for achieving a higher degree of accuracy in nowcasting the CCI is anticipated.

4 Explanatory analysis

The heatmap (Fig. 2) provides a visual representation of the strength and direction (correlation) of the relationship between the CCI and a set of variables over different time lags, ranging from 1 to 8 months. The most significant correlation is found with Inflation. It stands out with a strong negative correlation at a short 2-month lag (-0.67), indicating a potentially predictive relationship with the CCI. While sentiment indicators show variable correlations, Transformers_SI and Flair_SI at lag 5, along with Vader_SI at subsequent lags, display promising positive correlations. The Average_wage and Pension correlations suggest that higher wages and pensions may not always correlate with higher consumer confidence, potentially due to the complex nature of economic perception. Both Unemployment_without_seasonality and Unemployment_rate exhibit a negative correlation with CCI, more pronounced in the early lags.

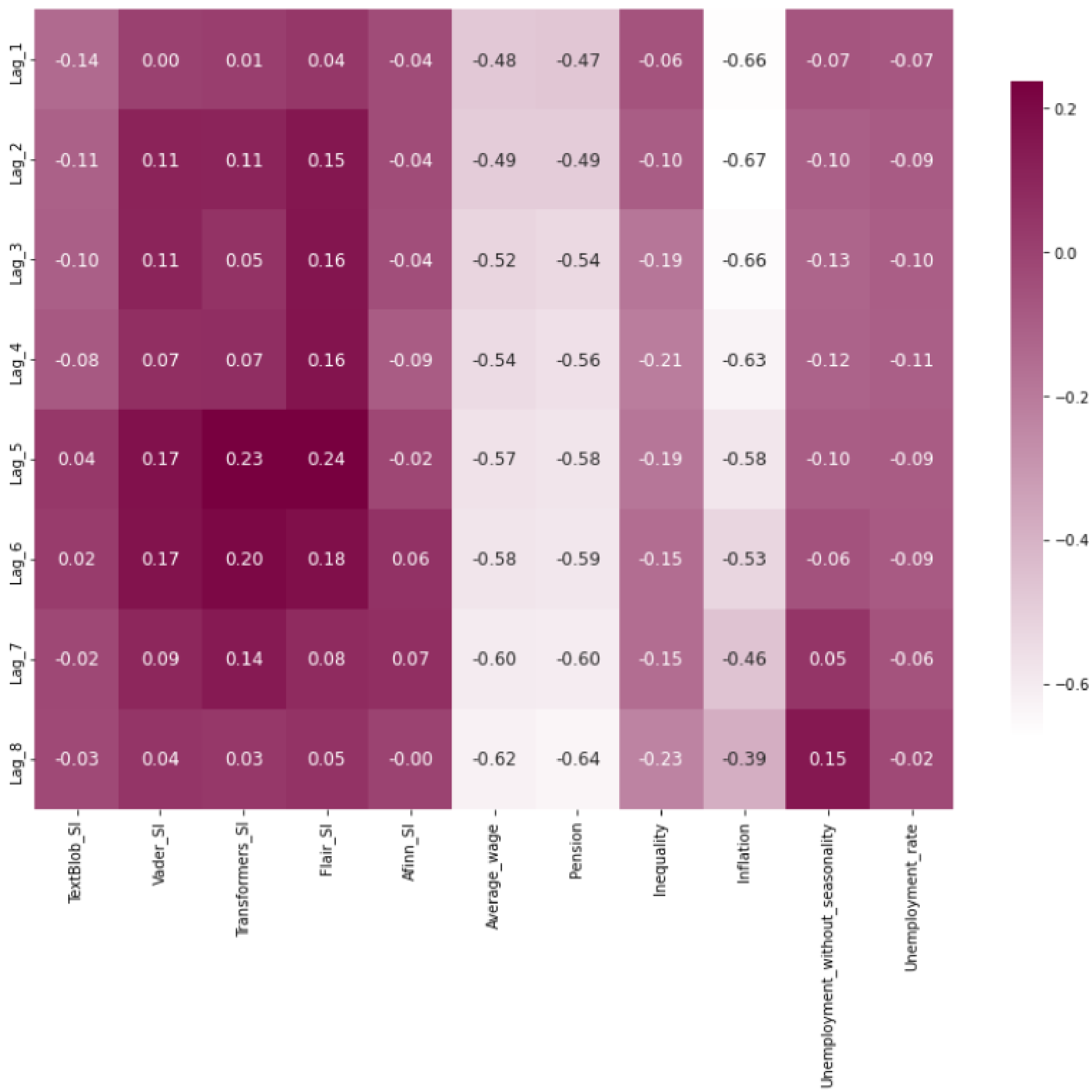


Figure 2: Correlations heatmap of CCI with lagged variables

Fig. 3 presents a visual overview of key economic indicators and sentiment indices. Notably, the CCI reflects a significant dip in April 2020, likely attributed to the initial impact of the COVID-19 pandemic, which officially began in March 2020. This event's economic ramifications are further illustrated by the subsequent peaks in unemployment indicators, which lag the CCI's trough by a few months, suggesting a delayed response in the labour market to the crisis.

Furthermore, following Russia's invasion of Ukraine in February 2022, a downward trend in the CCI can be observed, coinciding with a sharp rise in inflation. This inverse relationship between the CCI and inflation during this period echoes the patterns identified in the correlation heatmap (Fig. 2), underlining the potential impact of geopolitical tensions on consumer confidence and economic stability.

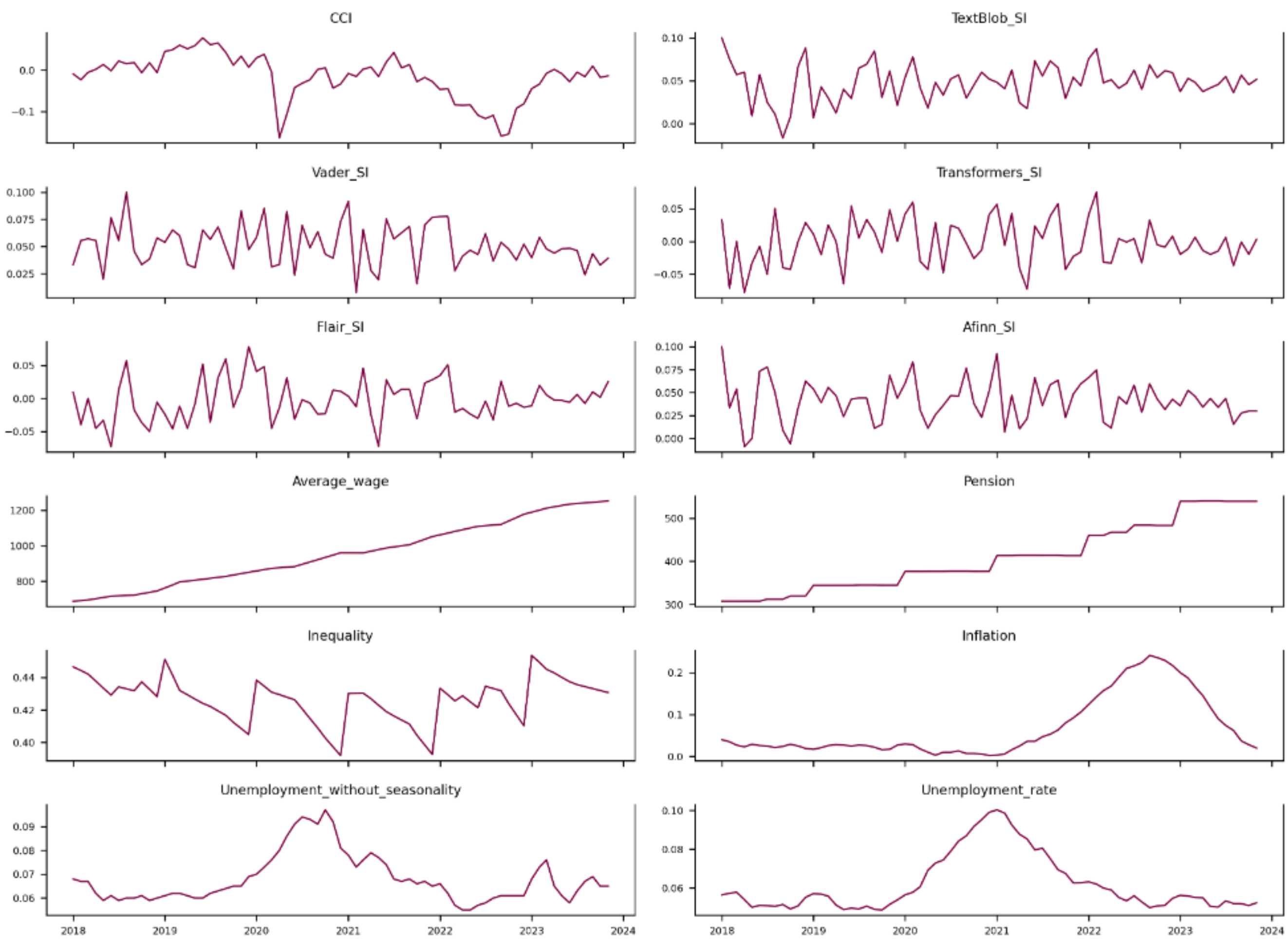


Figure 3: CCI, sentiment indices and key economic indicators

5 Evaluation metrics and rolling forecast approach for forecasting accuracy

To evaluate the accuracy of our forecasting models, we use an effective method called rolling forecast. This method works by predicting the next step and then updating the model with the latest information before making another prediction. It is like making a series of short-term guesses, one after the other, and getting better each time because we use the most recent data.

We evaluate the accuracy of our forecasting models using a set of statistical metrics:

- **Mean Absolute Error (MAE)**: MAE measures the average magnitude of errors in predictions, without considering their direction. It's given by the formula:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|, \tag{1}$$

  where y_i are the actual values, $\hat{y}_i$ are the predicted values, and n is the number of observations. MAE is a straightforward measure of prediction accuracy.

- **Mean Squared Error (MSE)**: MSE calculates the average squared difference between estimated and actual values. The formula for MSE is:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{2}$$

  MSE gauges the variance of forecast errors.

- **Root Mean Squared Error (RMSE)**: RMSE is the square root of MSE and measures the standard deviation of prediction errors. Its formula is:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}. \tag{3}$$

  RMSE is a scale-dependent accuracy measure often used in forecasting.

- **Akaike Information Criterion (AIC)**: AIC is a measure used for model selection that balances the complexity of the model with its fit to the data. The formula for AIC is:

$$\text{AIC} = 2k - 2\ln(L), \tag{4}$$

  where k is the number of parameters in the model and L is the maximum likelihood of the model. AIC is especially valuable for models with numerous parameters, like SARIMAX in our case, as it helps select a model that best explains the data with minimal complexity. A lower AIC value indicates a preferable model balance between accuracy and simplicity [25, 2].

Another one popular measure MAPE (Mean Absolute Percentage Error) is not used as it can be misleading when actual values are near zero, as with CCI values that oscillate around zero. The error metric can be disproportionately inflated, making MAPE unsuitable for our analysis [20].

For each of the forecasting models, we computed the MAE, MSE, and RMSE as key metrics to assess and compare their accuracy in forecasting the CCI. In the case of the SARIMAX model, we additionally calculated the AIC, which is particularly useful for comparing models with different numbers of parameters, providing a balance between model fit and simplicity.

6 Model development and implementation

This section explores 4 models to forecast CCI: SARIMAX, VECM, Random Forest, and XGBoost. Here we explain the reasons why these models were selected and discuss the selection of key exogenous variables and their impact on forecast accuracy. Additionally, we examine the practical application of these models, including their optimization and effectiveness in predicting CCI values. In this section, we aim to provide an understanding of the model development process.

6.1 SARIMAX

6.1.1 SARIMAX model fundamentals

The SARIMAX model, an extension of the well-established ARIMA model, is widely recognized for its effectiveness in time series analysis. As Manigandan et al. (2021) [30] notes, ARIMA is one of the most common methods in this field, and SARIMAX enhances its capabilities by accounting for seasonality and external factors.

The acronym SARIMAX stands for Seasonal AutoRegressive Integrated Moving Average with eXogenous variables. This model is denoted as SARIMAX(p,d,q)(P,D,Q)s, where 'p' represents the order of the autoregressive (AR) part, 'd' is the order of integration or differencing, and 'q' is the order of the moving average (MA) part. The seasonal components are captured by 'P' (seasonal AR order), 'D' (seasonal differencing order), 'Q' (seasonal MA order), and 's' indicates the frequency or number of observations per seasonal cycle.

A key feature of the SARIMAX model is the inclusion of 'X', signifying exogenous variables. These are external predictors or input variables that can influence the target variable being forecasted. Exogenous variables provide additional external information that can enhance the model's predictive capability [34].

6.1.2 Seasonal and trend decomposition using loess of the CCI

Seasonal and trend decomposition using loess (Locally Estimated Scatterplot Smoothing), in short, STL decomposition, is essential for revealing seasonality in time series, a key factor in applying models like SARIMAX effectively. It separates the data into trend, seasonal, and residual components, allowing us to visually discern seasonal patterns. This method, although not a statistical test, is enough to determine whether a series is seasonal to apply the appropriate model (in this case SARIMAX) for forecasting [34].

STL decomposition (Fig. 4) of the CCI series reveals distinct components that contribute to the overall behaviour of the time series. Observed data shows raw values, while the trend component reflects long-term progress, smoothing out short-term fluctuations. The STL decomposition reveals a noticeable seasonal pattern with noticeable cycles that indicate a strong seasonal influence. Although seasonality is clear, the variation in magnitude and timing of these cycles from year to year suggests that the pattern, while consistent, does not strictly follow a uniform structure.

The residuals representing the remaining variation after accounting for trends and seasonality look randomly distributed.

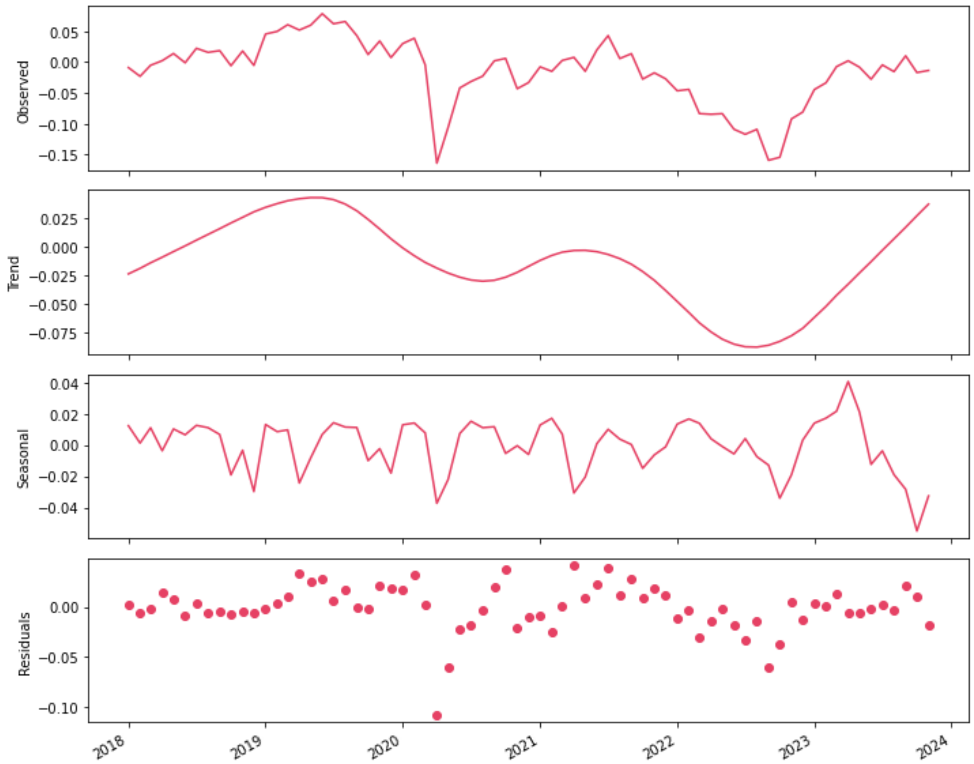


Figure 4: The STL decomposition of the CCI

The decomposition shows some seasonal effects with irregularities, supporting the decision to use the SARIMAX modelling approach.

6.1.3 Stationarity

Application of the SARIMAX model begins with ensuring data stationarity, which is a necessary condition for reliable time series forecasting. Using the Augmented Dickey-Fuller (ADF) test, each series in the data set is evaluated for stationarity. Also, the necessary order of differencing is determined to achieve this stationarity state. A summary table (Table 1) shows the variables and their respective differencing orders.

Variable	Differencing Order
TextBlob_SI	0
Vader_SI	1
Transformers_SI	0
Flair_SI	0
Afinn_SI	0
CCI	1
Average_wage	2
Pension	2
Inequality	2
Inflation	1
Unemployment_without_seasonality	1
Unemployment_rate	2

Table 1: Differencing Order Required for Stationarity

6.1.4 Selection of exogenous variables

The next step is to select exogenous variables that can improve the predictive ability of the model. This choice is based on Johansen's cointegration test to identify variables that have a long-run equilibrium relationship with the CCI. Johansen's test is typically used for time series that are I(1) (stationarity is obtained only after the first differencing) [22, 8]. Therefore, variables such as Vader_SI_diff_1, Inflation_diff_1, Unemployment_without_seasonality_diff_1, and 'CCI_diff_1' were selected for testing their cointegrated relationships (see Fig. 5). The test results suggest evidence of at least one and possibly more than one, but not more than two, cointegrating relationships among these four variables when considered together.

Lagged versions of selected variables are included in the model to capture temporal dependencies and lagged effects. Spearman's rank correlation test, a nonparametric method that assesses the strength and direction of a monotonic relationship between two variables, is used to systematically determine optimal lagged variables.

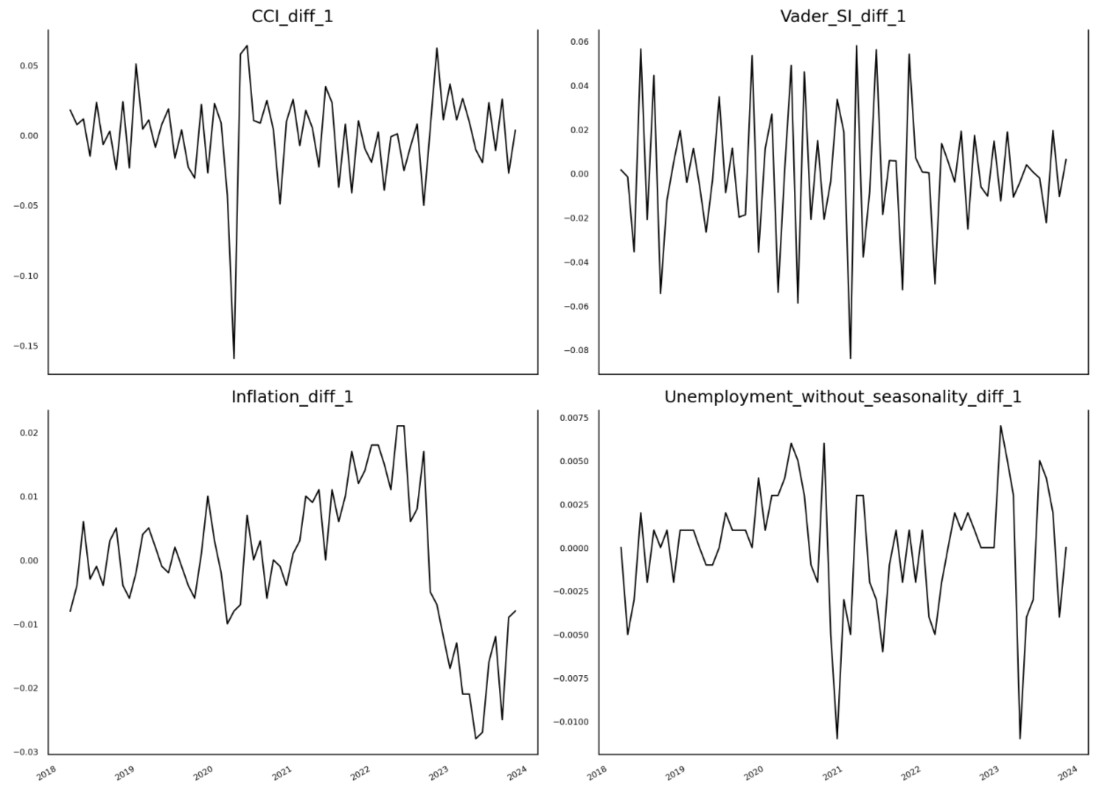


Figure 5: Differenced CCI and exogenous variables that have a long-run equilibrium relationship

The result of the correlation analysis is presented in a series of lines (Fig. 6), where each line shows the correlation of CCI_diff_1 with the exogenous variable at different lags. A lag of zero represents an unlagged variable, while positive lag values of 1, 2, 3, 6 and 12 months reflect the historical influence of exogenous variables on the CCI_diff_1.

It was decided to select Unemployment_without_seasonality_diff_1 with a six-month lag (0.22) and Inflation_diff_1 with its three-month lagged value (-0.29) which showed a moderate correlation with CCI_diff_1.

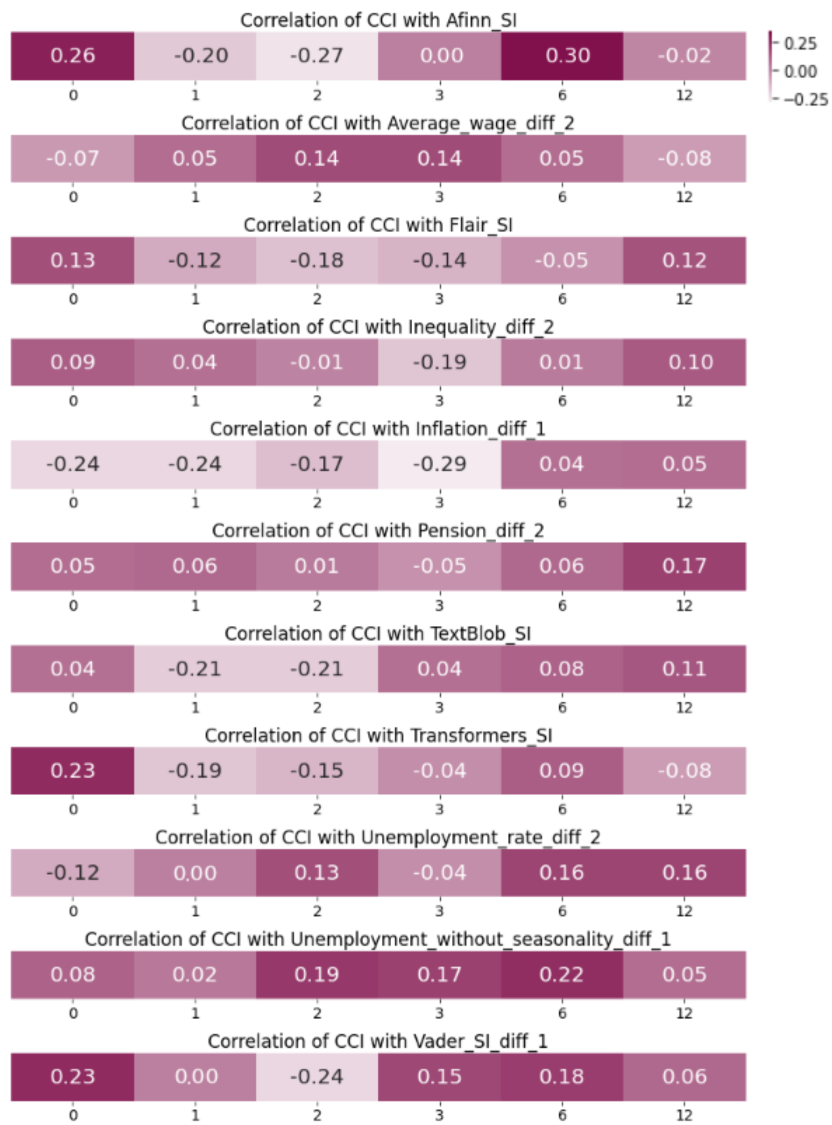


Figure 6: Correlation of CCI_diff_1 with exogenous variables and their lags

6.1.5 Optimal parameters selection using grid search

A grid search is performed to find the optimal parameters of the SARIMAX model. This involves iterating over a range of values for the non-seasonal and seasonal components of the SARIMAX(p,d,q)(P,D,Q)s model.

The order of integration d is set to 1 (series is stationary after being differenced once). The results of the ADF test on the seasonal component of CCI data show a very high negative ADF statistic and a p-value of 0.0. It means that the non-seasonal differencing (determined earlier as d = 1) is sufficient to make time series stationary, and no additional seasonal differencing is required for the SARIMAX model. Therefore D = 0. As data is recorded monthly, s is set to 12. The range up to 6 for 'p', 'q', 'P', and 'Q' is selected to capture the effects of recent trends and seasonal patterns up to a six-month horizon.

The optimal parameter combination for the SARIMAX model was selected based on MAE and AIC.

The MAE measures the model's predictive accuracy, while the AIC provides insight into the model's complexity, helping to avoid overfitting.

The combinations with parameters $p = 3$, $d = 1$, $q = 5$, $P = 0$, $D = 0$, and $Q = 3$ have an AIC of -222.366 and an MAE of 0.00812. This combination offers a good balance between a reasonably good AIC (while the lowest AIC was -232.469) and the lowest MAE. In this case, the emphasis is on MAE, because the model is used on an evolving dataset with new data points added monthly. Also, MAE is a comparative metric as it is used across other models in the study.

Therefore, the final model is SARIMAX(3,1,5)(0,0,3)12.

6.1.6 Residuals analysis

It is important to perform a residual analysis. Also, it is necessary to examine the Q-Q plot to ensure that it closely approximates a straight line, then apply the Ljung-Box test to see if the residuals are uncorrelated [34].

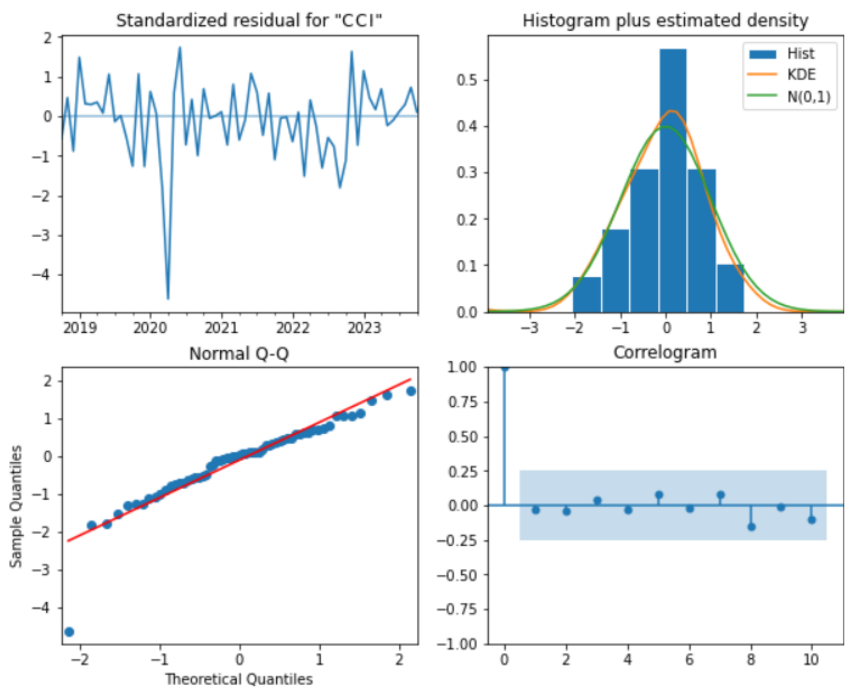


Figure 7: Diagnostic plots for testing normality and correlation of residuals

A diagnostic plots for testing normality and correlation of residuals is shown in Fig. 7. These plots are obtained after the final model fitting step. The standardized residuals plot shows no obvious systematic patterns or structural biases. The residuals oscillate around the zero line, indicating a well-fitted model that consistently captures the underlying trend of the time series without apparent bias. A close fit between the KDE (Kernel Density Estimate) and the normal distribution suggests that the residuals are approximately normally distributed. The alignment of data points with the theoretical line in a Normal Q-Q plot suggests that the residuals are normally distributed. The correlogram suggests that significant autocorrelation is present only at lag zero, indicating that the residuals are behaving as white noise.

The Ljung-Box test for the first 10 lags indicates all p-values are above the critical threshold of 0.05. It supports the null hypothesis that the residuals are independently distributed. Also, it confirms that the residuals are uncorrelated and the model's forecasts are reliable.

6.1.7 Forecasting and results

Data for the last 7 values (approximately 10% of data) is forecasted using a rolling forecast approach, predicting one-time step at a time to minimize the accumulation of forecast errors (Fig. 8). This iterative process ensures that each forecast incorporates the most recent actual data, enhancing the model's accuracy and responsiveness to new information.

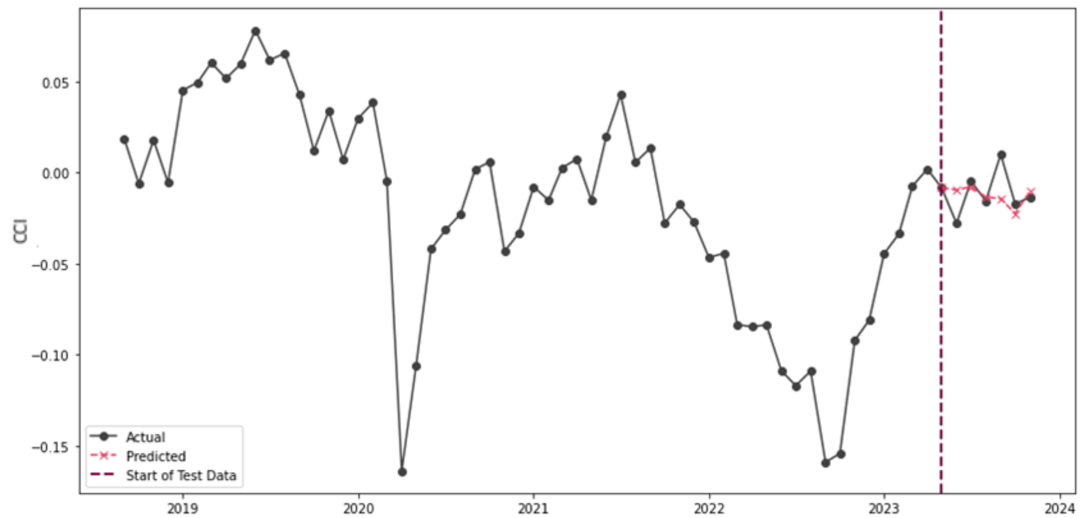


Figure 8: Actual vs predicted CCI SARIMAX(1,1,3)(1,0,3)12

6.2 VECM

6.2.1 VECM model fundamentals

The Vector Error Correction Model (VECM) is a model used for analyzing cointegrated time series [17]. It is particularly useful when dealing with non-stationary data that are integrated in the same order and cointegrated. Cointegration is essential for the VECM because it reflects the presence of long-run equilibrium relationships among variables. As noted by Engle et al. (1987) [15], cointegration indicates that certain sets of variables cannot significantly deviate from each other over time.

6.2.2 Model implementation

In developing the VECM for CCI forecasting, we used the main steps identified in the development of the SARIMAX model, taking into account the analogous data handling and variable selection requirements of both models. The main condition of both models is the stationarity of time series data. The Augmented Dickey-Fuller tests applied in the SARIMAX analysis to determine stationarity and to identify appropriate differences are also appropriate for the VECM model. The cointegration condition required for VECM is also verified during the analysis of the SARIMAX model.

The inclusion of exogenous variables in the SARIMAX model was based on Johansen's cointegration test and Spearman's rank correlation test. Based on these requirements, which are also relevant to the VECM model, we selected the same exogenous variables for the latter model.

Thus, we adopted a rolling forecasting approach, which is analogous to the method used in the SARIMAX model. We utilized the exogenous variables 'Vader_SI_diff_1', 'Inflation_diff_1', and 'Unemployment_without_seasonality_diff_1' in our model. Unlike the SARIMAX model, where we specified lagged versions of certain variables based on their correlation, the VECM inherently includes lagged versions of all exogenous variables. Therefore, to align with the nature of the VECM model, we chose a maximum lag of six (as for the SARIMAX model 'Unemployment_without_seasonality_diff_1 was lagged 6 months). However, the initial implementation of the model with these parameters resulted in a relatively high MAE of 0.14998, significantly larger than the MAE of 0.00812 achieved by the SARIMAX model.

6.2.3 Model optimization and results

To improve the model's performance, we explored a systematic approach of hyperparameter tuning. This involved iterating over different combinations of exogenous variables and lag orders to identify the configuration that yielded the best forecast accuracy based on the MAE. We found that using only 'Unemployment_without_seasonality_diff_1' as the exogenous variable with a maximum lag of 4 resulted in a more acceptable MAE of 0.01597. Although this MAE was not as low as that of the SARIMAX model, it represented a significant improvement over our initial VECM implementation.

Moreover, we confirmed the presence of cointegrating relationships between 'CCI_diff_1' and 'Unemployment_without_seasonality_diff_1', which are essential for the application of VECM. The Johansen cointegration test results indicated at least two cointegrating relationships among these variables. This finding underscores the long-term equilibrium relationship between these variables.

The VECM model's performance in forecasting the CCI is visually represented in the following figure (Fig. 9). It shows a comparison between the actual CCI values and those predicted by the VECM rolling forecasting method.

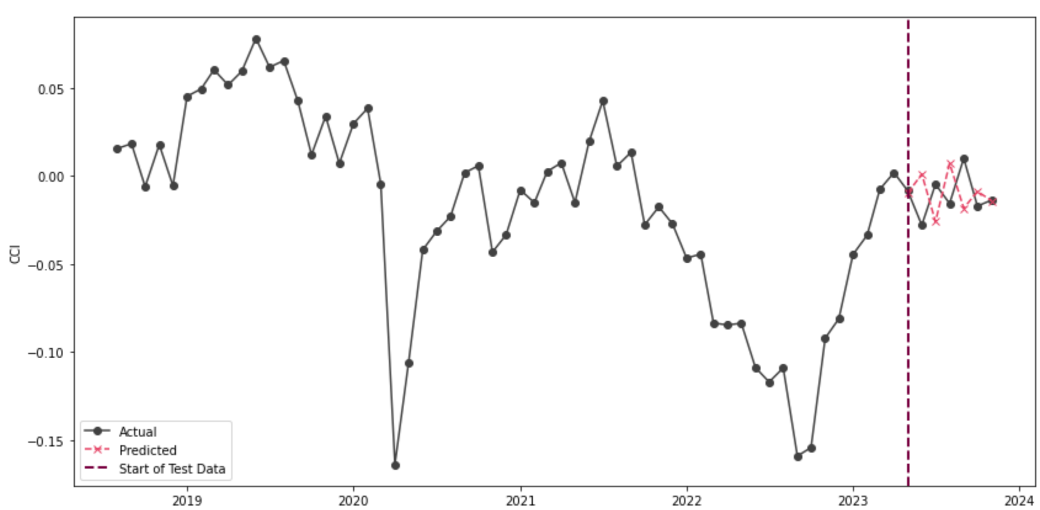


Figure 9: Comparison of actual vs. predicted CCI values using VECM rolling forecasting.

6.3 Random Forest

6.3.1 Random Forest model fundamentals

Random Forest (RF), known for its ability to handle complex datasets, is a powerful ensemble learning technique in machine learning. Breiman (2001) [7] explained that the Random Forest algorithm combines the outputs of numerous decision trees to get a single and more precise forecast. This helps avoid overfitting. Tyralis and Papacharalampous (2017) [38] explored RF for time series forecasting. They emphasised the importance of variable selection. Their study also examined how the number of lagged predictor variables affects the model's accuracy. It used measures such as MAE, MSE and MAPE to compare the performance of different forecasting methods. Cutler et al. (2012) [13] mentioned one of the main advantages of RF - evaluate feature importance. It can automatically identify the most influential exogenous variables called features in a dataset.

6.3.2 Model implementation

To predict the CCI for the last 7 months, we used the rolling forecast method. At first, the model used all variables from the dataset, including their lags from 1 to 12 months to catch possible annual influence.

Additionally, hyperparameter tuning was conducted as part of the model training process to refine the model's performance. This involved a systematic search for the best model parameters – such as the number of trees and their maximum depth – through a randomized search, which is a strategic approach to optimize model accuracy and minimize prediction errors.

After the initial model run (see Fig. 10a for the actual vs. predicted comparison), we get a quite low MAE = 0.01112 and the feature importance at each forecasting step (see Fig. 13). This helped identify the most influential features and their respective lags. Based on this analysis, the model was refined by selecting the top features and their specific lags for a more optimized forecasting model. The optimization process then is described in the following section.

6.3.3 Model optimization

We gained insights from the initial feature importance analysis in the model optimisation phase. The CCI lagged one month consistently held the most significant weight across multiple forecasting steps. This finding underscores the pivotal role that the CCI from the previous month plays in predicting the value for the ensuing month. Additionally, Inflation and Pension both lagged for 3 months and emerged as recurrently significant.

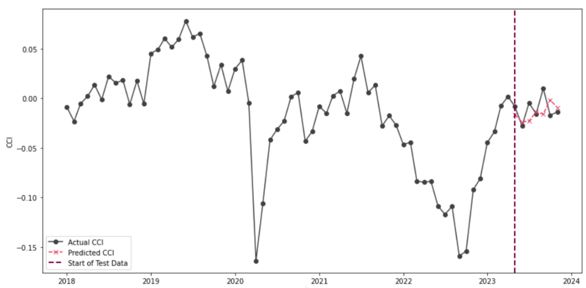
Further analysis revealed that incorporating sentiment indicators like TextBlob_SI in both unlagged and lagged forms (specifically, current, one-month lagged, and seven-month lagged) could enhance predictive performance. Applying this refined set of features resulted in a lower MAE, down to 0.01049 from the initial 0.01158 (see Fig. 10b). Feature importance after this step is shown in Fig. 14.

Taking it a step further, we incorporated rolling averages. This approach was predicated on the hypothesis that smoothing out short-term fluctuations could reveal underlying trends more pertinent to the forecasting task. Adjusting the rolling windows to 4 months for Inflation and CCI, and 5 months for Pension, brings down the MAE to 0.00993 (see Fig. 10c, Fig. 15).

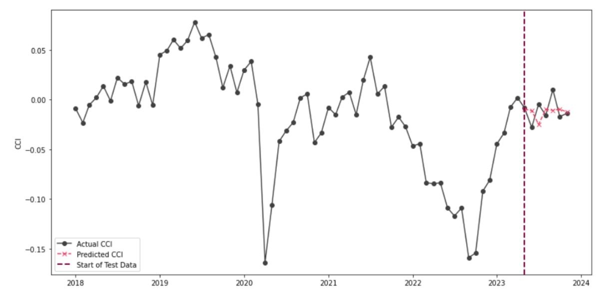
6.3.4 Model evaluation and results

The final CCI forecasting model for the past 7 months included a combination of non-lagged, lagged and rolling average features to optimize forecasting accuracy. The model features were CCI lagged by 1 month, TextBlob_SI in its current form, as well as lagged by 1 and 7 months, Inflation, and Pension lagged by 3 months. The rolling averages applied were 4 months for both Inflation and CCI, and 5 months for Pension.

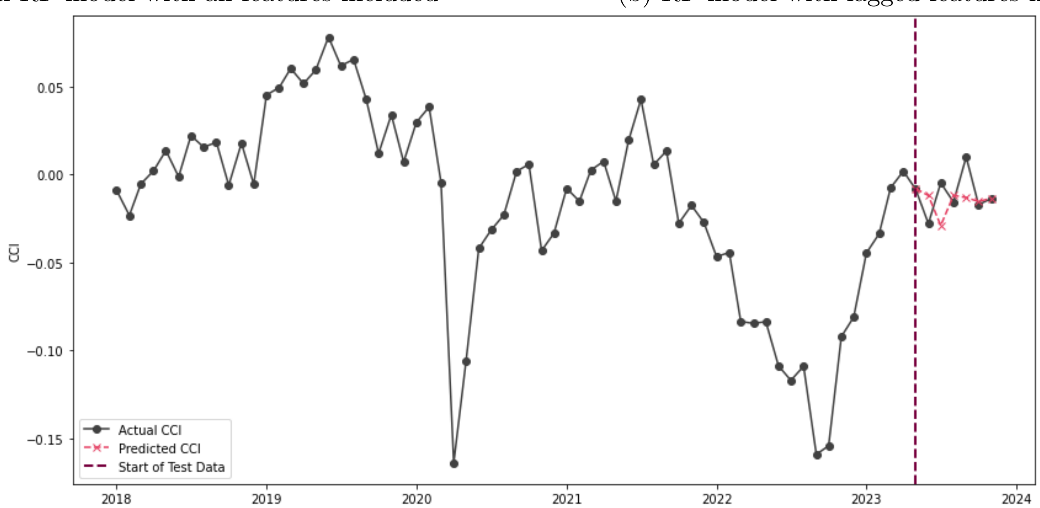
The evaluation of the model's performance and the illustrative results from each run can be observed in Fig. 10, where the graphical representations provide a comparison between the predicted and actual values of the CCI.



(a) Innitial RF model with all features included



(b) RF model with lagged features included



(c) Final RF model with lagged features and rolling averages included

Figure 10: RF models

6.4 XGBoost

6.4.1 XGBoost model fundamentals

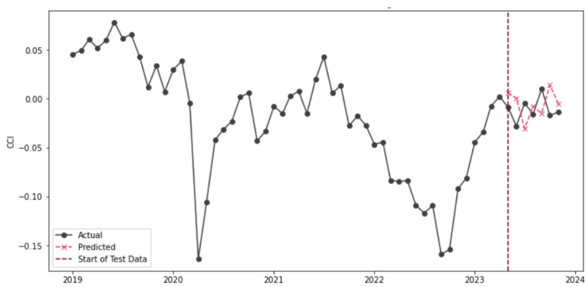
XGBoost (eXtreme Gradient Boosting) is an advanced implementation of gradient boosting algorithms. It has become a very popular tool for machine learning competitions. It can reveal complex non-linear patterns and common features of time series data. By analyzing time series, XGBoost builds an ensemble of regression trees using features such as early stopping to optimize training time and performance. A key feature of XGBoost is its ability to evaluate feature importance, which indicates how much each feature contributes to the prediction. This insight is particularly useful for identifying and eliminating non-contributing features from the model [9, 40].

6.4.2 Model implementation, optimization and results

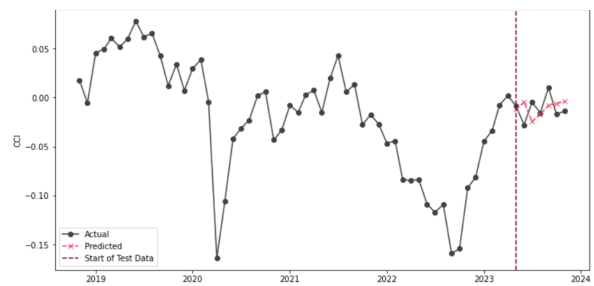
We employed a methodology akin to our approach with the Random Forest model. Initially, to forecast the last 7 months of CCI, we fed the model with all features from our dataset, incorporating their lagged values up to 12 months. Therefore, the MAE was relatively high at 0.01999 (see Fig. 11a for the actual vs. predicted comparison).

The feature importance graph (Fig. 16) allowed us to identify and focus on the most impactful features. Notably, we observed that including the CCI lagged by one month, the current value of 'Average_wage', 'Transformers_SI' shifted by two months, and 'Unemployment_without_seasonality' with a lag of 10 months enhanced the model's performance. To further investigate the influence of this latter variable, we also experimented with a shorter delay of just one month and with the current value. Our experimentation revealed that applying a 1-month lag and incorporating the current value (0-month lag) of this variable significantly improved the model's precision. Consequently, we chose to include only these short-term lags – 0 and 1 month – in our final model, which resulted in a lower MAE of 0.01396, as demonstrated in Fig. 11b, Fig. 17. This decision was based on the clear evidence that shorter lags provided a better forecast than the initially considered 10-month lag.

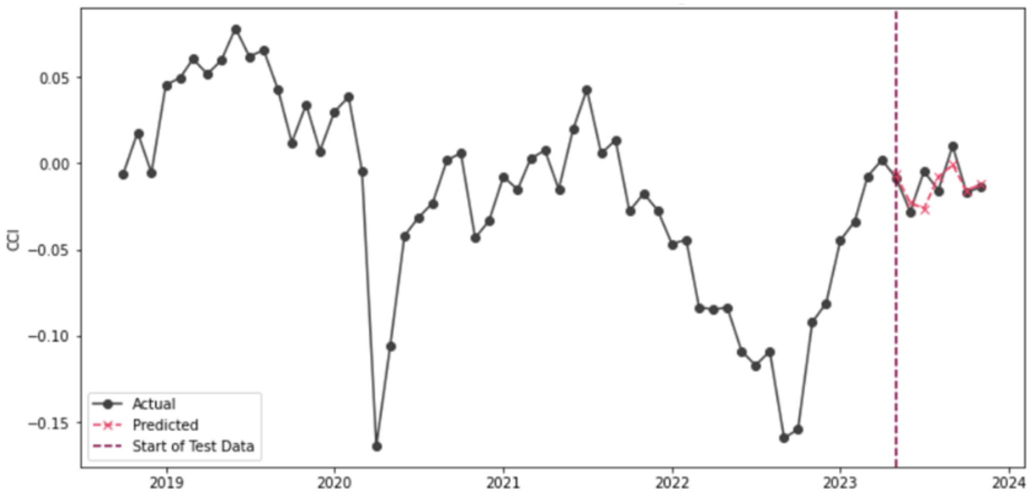
Also, we introduced rolling averages for key variables, utilizing a Rolling Window Grid Search for optimization. This approach involved testing various window sizes to find the best fit for each feature. For the CCI, a rolling window of 6 periods proved most effective. The average wage data benefited from a larger 9-period window, capturing longer-term trends. Sentiment analysis from the Transformers model and seasonally adjusted unemployment figures both showed optimal results with a 4-period window, allowing the model to quickly adapt to recent changes. Incorporating these specific rolling windows significantly improved our model's accuracy, reducing the MAE to 0.00713 (see Fig. 11c, Fig. 18).



(a) Innitial model with all features included



(b) XGBoost model with lagged features included



(c) Final XGBoost model with lagged features and rolling averages included

Figure 11: XGBoost models

The improvements of our model after each optimization step are illustrated in Fig. 11, which shows a comparison of actual vs predicted values and demonstrates the increased accuracy achieved by systematically tunning efforts.

7 Conclusions

7.1 Summary of findings

In this work, we aimed to predict CCI. We explored 4 different models incorporating data from social media and administrative sources. The selection and integration of exogenous variables, taking into account their temporal influence through lags and rolling averages, helped to increase the models' accuracy. Table 2 shows the specific external variables used in each model.

The final models for SARIMAX, VECM, Random Forest, and XGBoost are presented in Fig. 12. This figure provides a visual comparison of the actual CCI values against the predictions generated by each model.

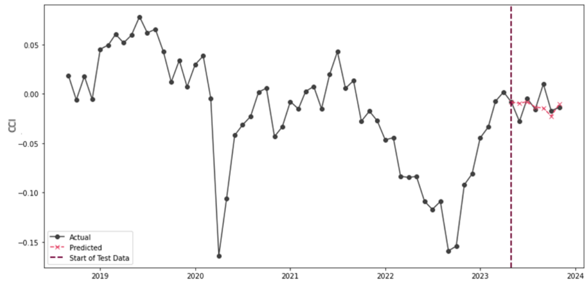
Model Name	**Exogenous variables (features)**
SARIMAX	Vader_SI_diff_1, Inflation_diff_1, Unemployment_without_seasonality_diff_1 (lag 6 months), Inflation_diff_1 (lag 3 months)
VECM	Unemployment_without_seasonality_diff_1 (lags up to 4 months) [1]
Random Forest	CCI (lag 1 month), TextBlob_SI (current, lag 1 and 7 months), Inflation (lag 3 months), Pension (lag 3 months), Rolling averages for Inflation (4 periods), CCI (4 periods), and Pension (5 periods)
XGBoost	CCI (lag 1 month), Average_wage (current), Transformers_SI (lag 2 months), Unemployment_without_seasonality (current and lag 1 month), Rolling averages for CCI (6 periods), Average_wage (9 periods), Unemployment (4 periods)

Table 2: Exogenous variables (features) in final forecasting models

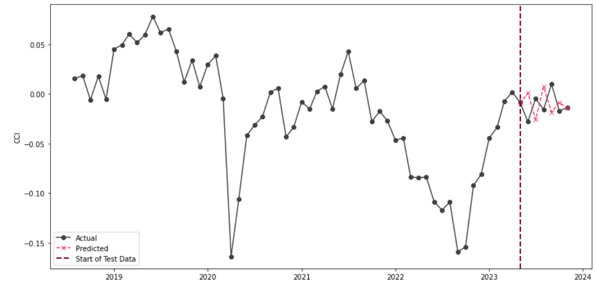
The forecasting capabilities of each model were measured with Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and the Akaike Information Criterion (AIC) for SARIMAX. These metrics are presented in Table 3.

Based on the accuracy metrics (Table 3) and visual comparisons shown in the graphs (Fig. 12), XGBoost demonstrated better performance compared to the other models. The SARIMAX model followed closely, with a strong predictive ability, although slightly less accurate than XGBoost. The Random Forest and VECM models, while still valuable for their predictive insights, did not match the level of accuracy achieved by XGBoost and SARIMAX.

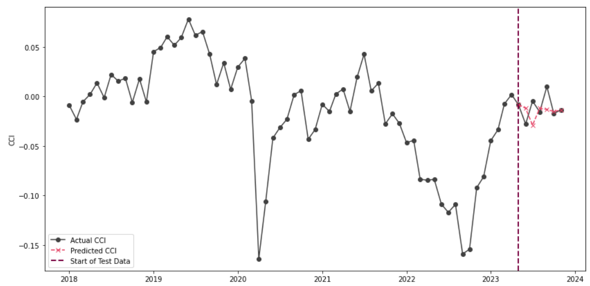
[1] In the VECM model, $max\_lags = 4$ applies to all variables in the system, meaning that up to 4 lagged values of each variable, including the variable CCI_diff_1, are included in the model.



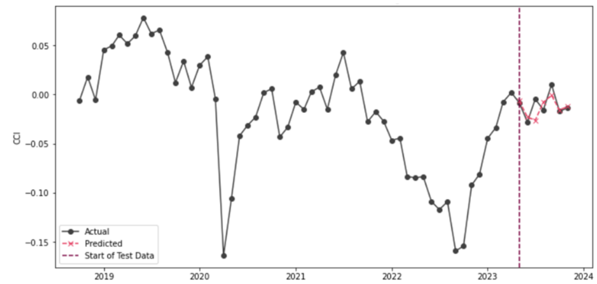
(a) SARIMAX final model



(b) VECM final model



(c) Random forest final model



(d) XGBoost final model

Figure 12: Final models for CCI forecasting

Model Name	**MAE**	**MSE**	**RMSE**
SARIMAX	0.00812	0.000066	0.00812
VECM	0.01597	0.000255	0.01597
Random Forest	0.00993	0.000098	0.00993
XGBoost	0.00713	0.000051	0.00713

Table 3: Forecasting errors of each model

7.2 Future work

The study offers a solid foundation for predicting of the Consumer Confidence Index (CCI) using social media and administrative data. However, it also opens up several prospects for future research. One such avenue is the exploration of advanced methodologies not yet explored in this thesis, such as Long Short Term Memory (LSTM) networks. These differ from the models currently employed. The thesis combines the econometric precision of SARIMAX and VECM models with the predictive capabilities of machine learning algorithms like XGBoost and Random Forest, which do not utilize the sequential memory characteristic of recurrent neural network (RNN) models like LSTM. Additionally, future research could benefit from incorporating and analyzing data from a variety of sources, including other economic indicators and Google Trends data to enhance the comprehensiveness and accuracy of the CCI prediction models.

References

[1] R. Aishwarya, C. Ashwatha, A. Deepthi, and J. B. Raja. A novel adaptable approach for sentiment analysis. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 5(2), 2019.

[2] H. Akaike. Factor analysis and AIC. *Psychometrika*, 52:317–332, 1987.

[3] Alan Akbik, Tanja Bergmann, Duncan Blythe, Kashif Rasul, Stefan Schweter, and Roland Vollgraf. Flair: An easy-to-use framework for state-of-the-art nlp. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics (demonstrations)*, pages 54–59, 2019.

[4] M. Ashraf, A. A. Raza, and M. Ishaq. A novel approach of social media analytics for predicting national consumer confidence index. *Bulletin of Business and Economics (BBE)*, 11(2):220–234, 2022.

[5] E. Baldacci, B. Braaksma, A. G'alvez, K. Giannakouris, B. G. Olmos, P. Rivi'ere, M. Scannapieco, A. Vermeer, and V. Vertanen. Innovation during the covid-19 crisis: Why it was more critical for official statistics than ever. *Statistical Journal of the IAOS*, (Preprint):1–14, 2022.

[6] J. van den Brakel, E. Söhler, P. Daas, and B. Buelens. Social media as a data source for official statistics; the dutch consumer confidence index. *Survey Methodology*, 43(2):183–210, 2017.

[7] L. Breiman. Random Forests. *Machine learning*, 45:5–32, 2001.

[8] E. Catanese, M. Scannapieco, M. Bruno, and L. Valentino. Natural language processing in official statistics: The social mood on economy index experience. *Statistical Journal of the IAOS*, (Preprint):1–9, 2022.

[9] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[10] L. Corti, M. Zanetti, G. Tricella, and M. Bonati. Social media analysis of Twitter tweets related to asd in 2019–2020, with particular attention to covid-19: topic modelling and sentiment analysis. *Journal of big Data*, 9(1):113, 2022.

[11] R. Curtin. Consumer sentiment surveys: worldwide review and assessment. *Journal of business cycle measurement and analysis*, 2007(1):7–42, 2007.

[12] R. Curtin, S. Presser, and E. Singer. The effects of response rate changes on the index of consumer sentiment. *Public opinion quarterly*, 64(4):413–428, 2000.

[13] D. R. Cutler, T. C. Edwards Jr, K. H. Beard, A. Cutler, K. T. Hess, J. Gibson, and J. J. Lawler. Random forests for classification in ecology. *Ecology*, 88(11):2783–2792, 2007.

[14] S. K. Demirel and S. Artan. The causality relationships between economic confidence and fundamental macroeconomic indicators: Empirical evidence from selected european union countries. *International Journal of Economics and Financial Issues*, 7(5):417, 2017.

[15] R. F. Engle and C. WJ. Granger. Co-integration and error correction: representation, estimation, and testing. *Econometrica: journal of the Econometric Society*, pages 251–276, 1987.

[16] Eurostat, the statistical office of the European Union. Business and consumer surveys (source: Dg ecfin) (ei_bcs). https://ec.europa.eu/eurostat/cache/metadata/en/ei_bcs_esms.htm, 2023. Metadata last posted 30/05/2023.

[17] S. Hajifar, H. Sun, F. M. Megahed, L. A. Jones-Farmer, E. Rashedi, and L. A. Cavuoto. A forecasting framework for predicting perceived fatigue: Using time series methods to forecast ratings of perceived exertion with features from wearable sensors. *Applied Ergonomics*, 90:103262, 2021.

[18] E. Hargittai and E. Litt. The tweet smell of celebrity success: Explaining variation in Twitter adoption among a diverse group of young adults. *New media & society*, 13(5):824–842, 2011.

[19] Clayton Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Proceedings of the international AAAI conference on web and social media*, volume 8, pages 216–225, 2014.

[20] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International journal of forecasting*, 22(4):679–688, 2006.

[21] T. U. Islam and M. N. Mumtaz. Consumer confidence index and economic growth: An empirical analysis of EU countries. *EuroEconomica*, 35(2), 2016.

[22] S. Johansen. *Likelihood-based inference in cointegrated vector autoregressive models*. OUP Oxford, 1995.

[23] G. Katona and R. Likert. Relationship between consumer expenditures and savings: The contribution of survey research. *The Review of Economics and Statistics*, 28(4):197–199, 1946.

[24] S. Kemp. Digital 2023: Lithuania. https://datareportal.com/reports/digital-2023-lithuania, 2023. Published on February 13, 2023.

[25] Y. W. Lee, K. G. Tay, and Y. Y. Choy. Forecasting electricity consumption using time series model. *International Journal of Engineering & Technology*, 7(4.30):218–223, 2018.

[26] State Data Agency (Statistics Lithuania). Consumer survey. https://osp.stat.gov.lt, 2023. Metadata last posted 2023-05-29.

[27] B. Liu. *Sentiment analysis and opinion mining*. Springer Nature, 2022.

[28] J. Logemann. Measuring and managing expectations: Consumer confidence as an economic indicator, 1920s–1970s. *Futures Past. Economic Forecasting in the 20th and 21st Century*, 2020.

[29] S. Loria et al. textblob documentation. *Release 0.16*, page 73, 2020.

[30] P. Manigandan, M. S. Alam, M. Alharthi, U. Khan, K. Alagirisamy, D. Pachiyappan, and A. Rehman. Forecasting natural gas production and consumption in united states-evidence from sarima and sarimax models. *Energies*, 14(19):6021, 2021.

[31] E. Mueller. Ten years of consumer attitude surveys: Their forecasting record. *Journal of the American Statistical Association*, 58(304):899–917, 1963.

[32] F. Å. Nielsen. A new anew: Evaluation of a word list for sentiment analysis in microblogs. *arXiv preprint arXiv:1103.2903*, 2011.

[33] L. Nowzohour and L. Stracca. More than a feeling: Confidence, uncertainty, and macroeconomic fluctuations. *Journal of Economic Surveys*, 34(4):691–726, 2020.

[34] M. Peixeiro. *Time series forecasting in Python*. Simon and Schuster, 2022.

[35] S. Stieglitz, M. Mirbabaie, B. Ross, and C. Neuberger. Social media analytics–challenges in topic discovery, data collection, and data preparation. *International journal of information management*, 39:156–168, 2018.

[36] The Conference Board. Consumer confidence survey technical note – may 2021. https://www.conference-board.org/, May 2021.

[37] A. Tumasjan, T. Sprenger, P. Sandner, and I. Welpe. Predicting elections with Twitter: What 140 characters reveal about political sentiment. In *Proceedings of the international AAAI conference on web and social media*, volume 4, pages 178–185, 2010.

[38] H. Tyralis and G. Papacharalampous. Variable selection in time series forecasting using Random Forests. *Algorithms*, 10(4):114, 2017.

[39] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

[40] N. Woloszko. Adaptive trees: a new approach to economic forecasting. 2020.

Appendices

Išlaidos	Išlaidas	Išlaidų	Išleidžiu	Išleidžiame
Išleisti	Kaštai	Santaupos	Sutaupau	Santaupų
Santaupas	Taipome	Taupau	Taupyti	Taupymas
Sutaupome	Sutaupyti	Pirkti	Pirkinys	Pirkiniai
Pirkinių	Perka	Perku	Pirksite	Perkame
Nuperku	Nusiperku	Bedarbiai	Bedarbis	Bedarbių
Bedarbiams	Bedarbius	Nedarbas	Nedirbu	Nedirbti
Darbas	Darbai	Darbą	Darbu	Dirbti
Dirbu	Ekonomika	Ekonominė padėtis	Finansinė padėtis	Finansai
Finansus	Finansinis	Finansų	Pinigai	Pinigus
Pinigų	Pinigais	Kainuoti	Kainuoja	Kaina
Kainos	Kainose	Kainų	Kainų pokyčiai	Defliacija
Infliacija	Infliacijai	Infliuoti	Infliacijos	Atlyginimas
Atlyginima	Atlyginimai	Alga	Algos	Pajamos
Pajamomis	Pajamų	Palūkanų normos	Palūkanų	Palūkanos
Palūkanas	Biudžetas	Biudžetą	Biudžetui	Biudžetu
BVP	Bendras vidaus produktas	Rinka	Mokesčiai	Mokesčius
Mokestis	Sumokėti	Mokėjimas	Pensija	Pensijos
Pensijas	Skola	Skolintis	Skolinti	Skolinu
Skolinuosi	Pasiskolinti	Įsiskolinimas	Įsiskolinu	Įsiskolinti
Namų ūkis				

Table 4: List of economic keywords

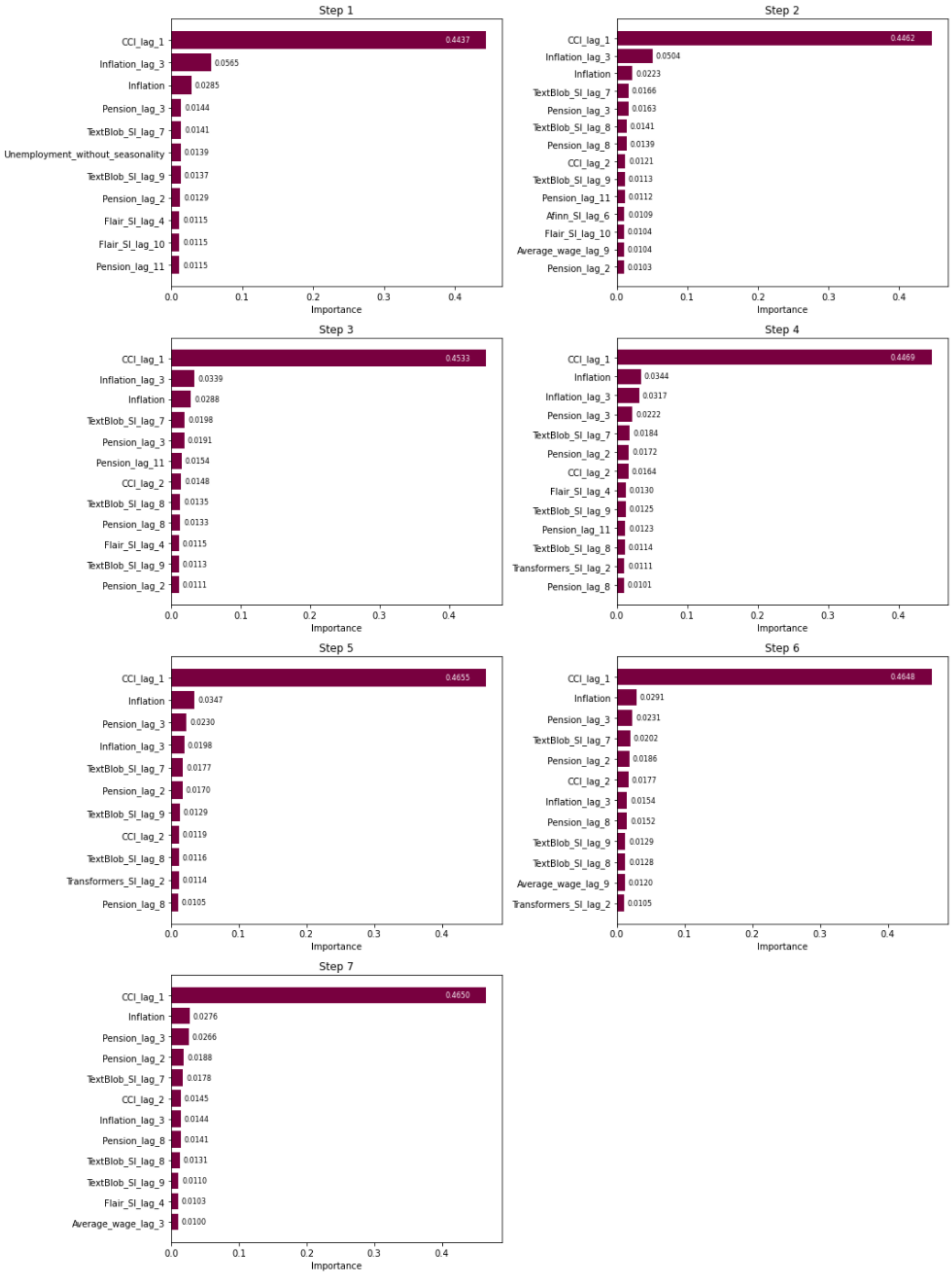


Figure 13: Feature importance for Random Forest model using all features

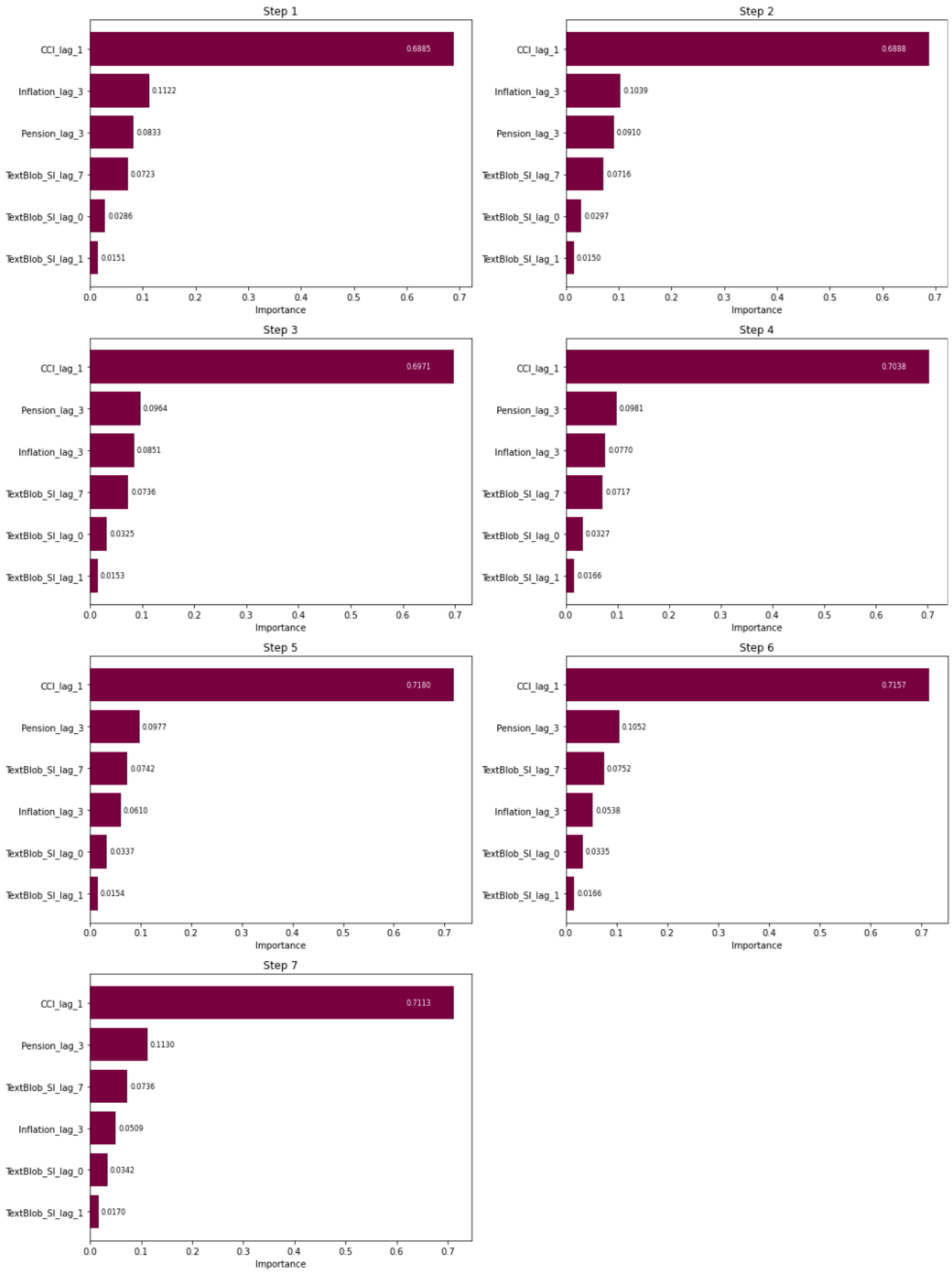


Figure 14: Feature importance for Random Forest model using selected features and their lags

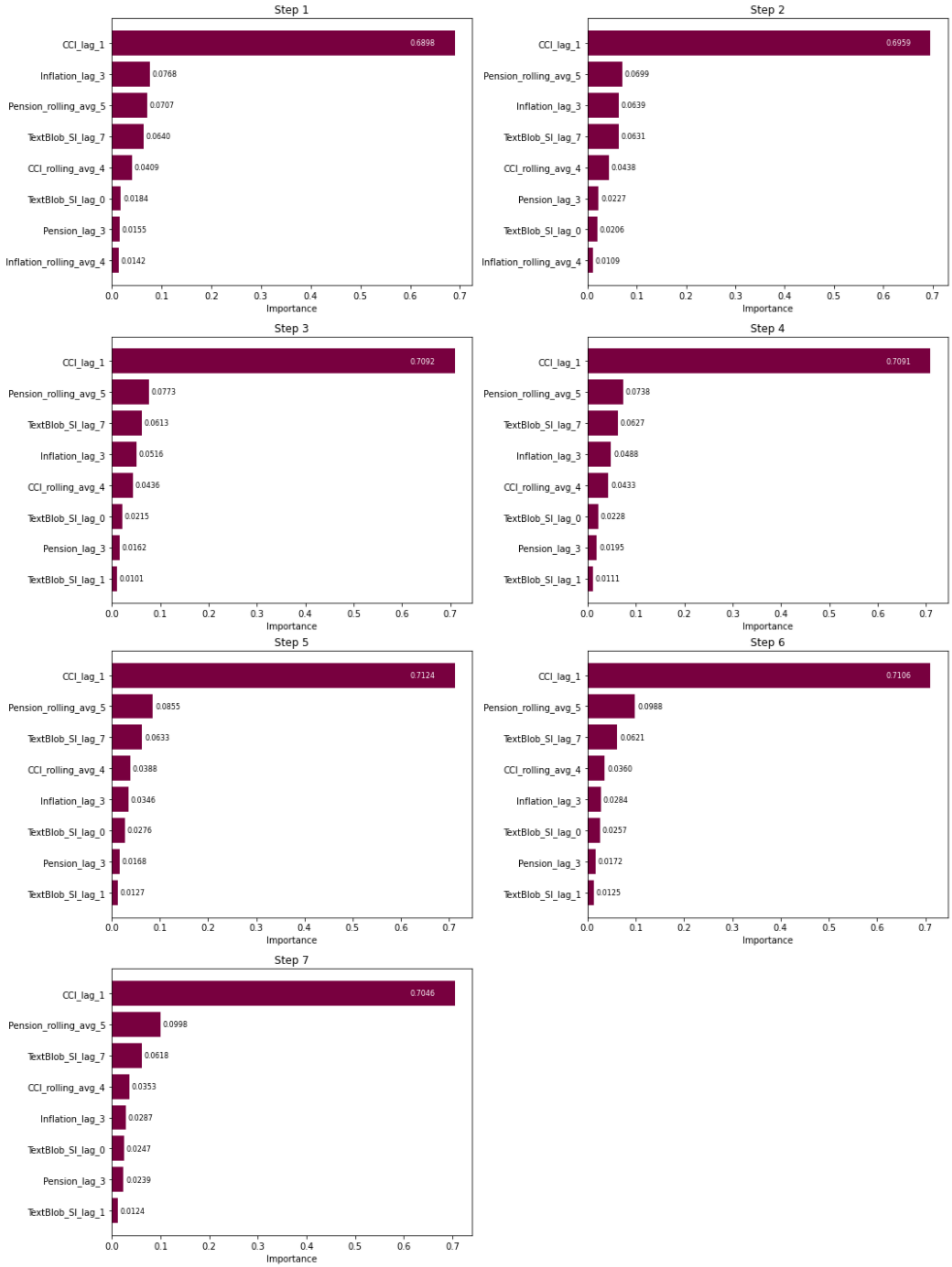


Figure 15: Feature importance for Random Forest model using selected features, their lags and rolling averages

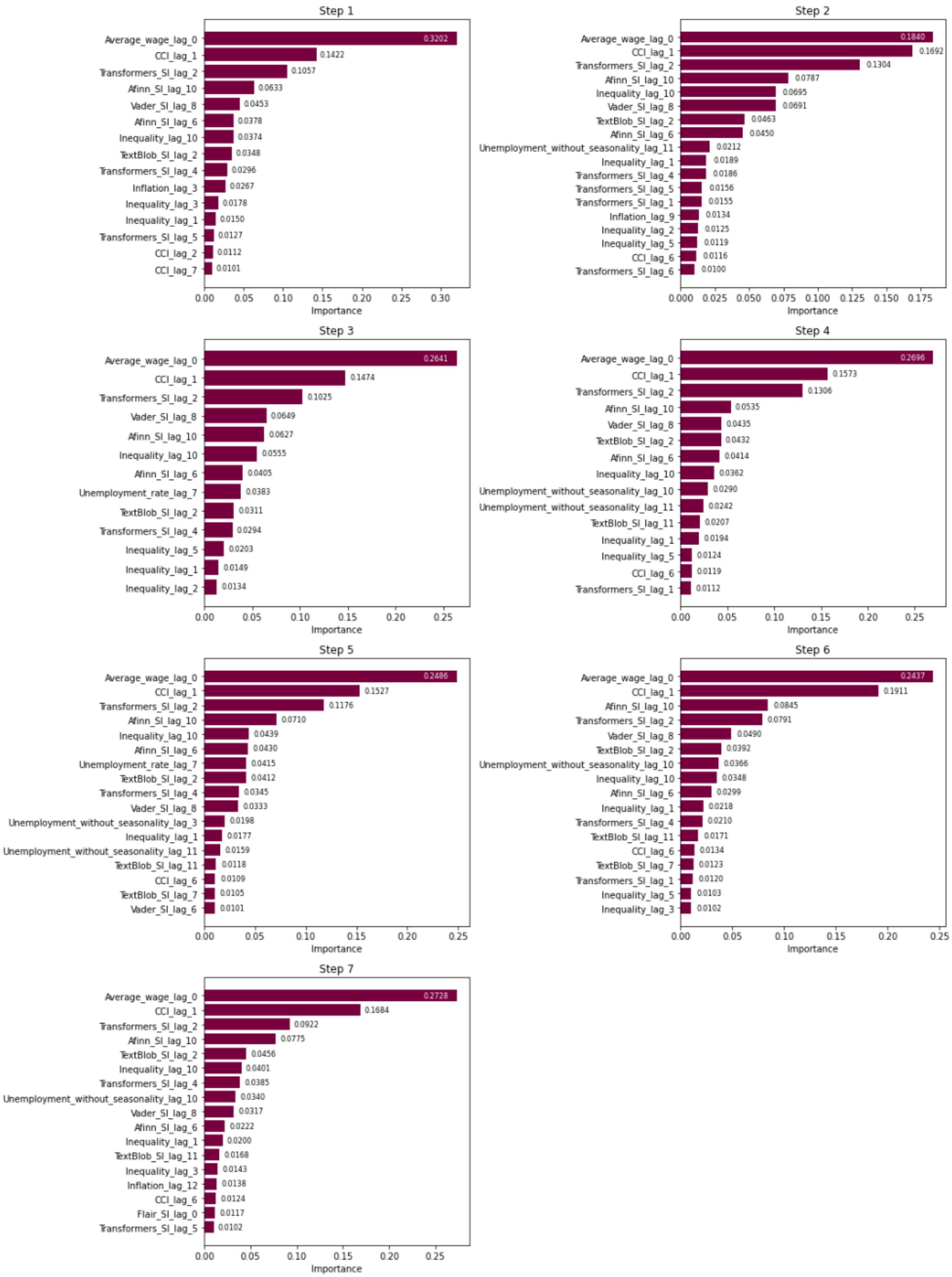


Figure 16: Feature importance for XGBoost model using all features

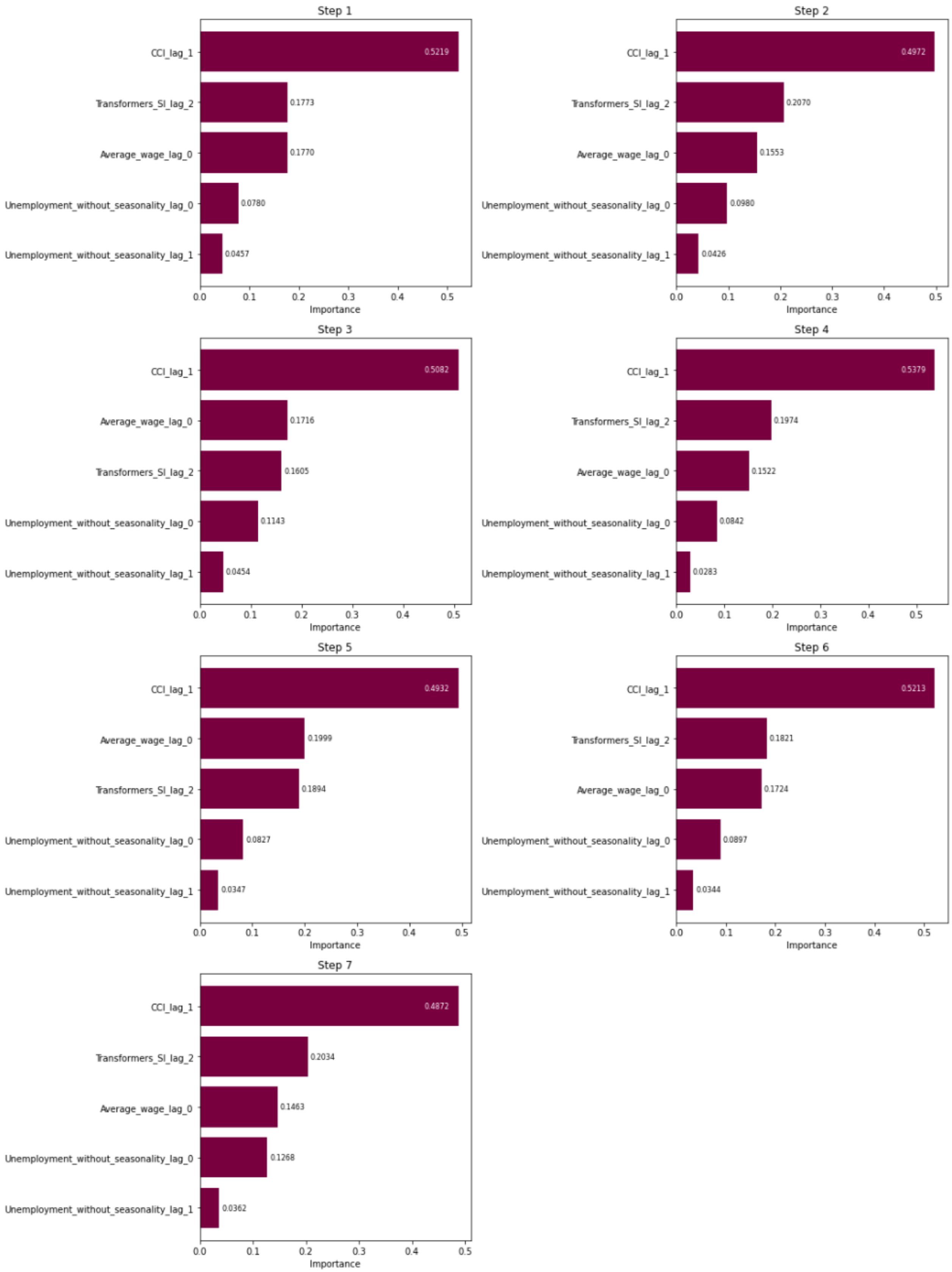


Figure 17: Feature importance for XGBoost model using selected features and their lags

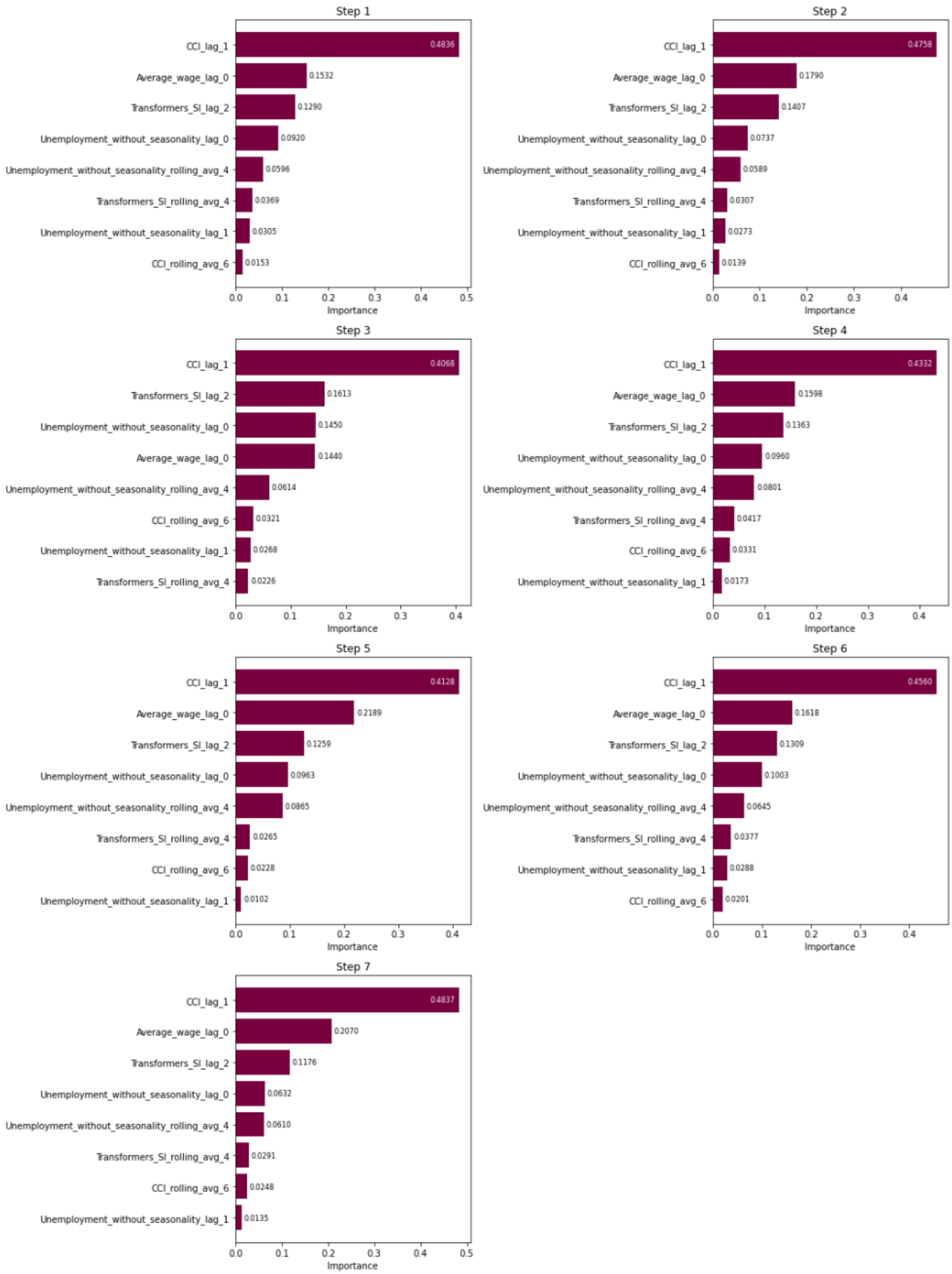


Figure 18: Feature importance for XGBoost model using selected features, their lags and rolling averages

Here we provide the main code parts used.

```
#——Libraries————————————————————————
# Data Handling
import pandas as pd
import numpy as np
import os
import os.path
import time
import datetime as dt
from datetime import datetime

# Text and Language Processing
from unidecode import unidecode
from langdetect import detect
import re

# Twitter API Interaction
import tweepy

# Sentiment Analysis Libraries
from textblob import TextBlob
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from transformers import pipeline
import flair
import stanza
from afinn import Afinn

# File Handling
import xlsxwriter
import openpyxl

# HTTP and SSL for API Interaction
import requests
from requests.adapters import HTTPAdapter
from urllib3.util.ssl_ import create_urllib3_context
import ssl

# Statistical and Forecasting Models
from statsmodels.tsa.seasonal import STL
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.vector_ar.vecm import VECM
from statsmodels.tsa.stattools import adfuller
import scipy.stats as stats

# Machine Learning Models and Evaluation
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error,
    mean_absolute_percentage_error
```

```
import xgboost as xgb

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Miscellaneous
from itertools import product
from math import sqrt
from pprint import pprint

#---Weekly scraping---------------------------------------------------
client = tweepy.Client(bearer_token=bearer_token)
# tweet_fields
tweet_fields = ['attachments',
                'author_id',
                'context_annotations',
                'conversation_id',
                'created_at',
                'edit_controls',
                'edit_history_tweet_ids',
                'entities',
                'geo',
                'id',
                'in_reply_to_user_id',
                'lang',
                'possibly_sensitive',
                'public_metrics',
                'referenced_tweets',
                'reply_settings',
                'source',
                'text',
                'withheld'
               ]

user_fields = ['created_at',
               'description',
               'entities',
               'id',
               'location',
               'pinned_tweet_id',
               'profile_image_url',
               'protected',
               'public_metrics',
               'url',
               'verified',
               'verified_type',
               'withheld'
              ]
```

```
place_fields = ['contained_within',
                'country',
                'country_code',
                'full_name',
                'geo',
                'id',
                'name',
                'place_type'
               ]

expansions = ['author_id',
              'geo.place_id',
              'attachments.media_keys'
             ]

def search_tweets(query, tweet_fields, user_fields, place_fields, expansions,
    start_time, max_results):
    """
    This function searches for recent tweets based on various parameters.

    Parameters:
    - query: The search query to find tweets.
    - tweet_fields: Fields to include in the tweet objects.
    - user_fields: Fields to include in the user objects.
    - place_fields: Fields to include in the place objects.
    - expansions: Expansions to include additional objects in the payload.
    - start_time: The oldest UTC timestamp from which the tweets will be provided.
    - max_results: The max number of search results returned by a single API call.

    Returns:
    A list of tweets that match the search criteria.
    """

    tweets = client.search_recent_tweets(query = query,
                                         tweet_fields = tweet_fields,
                                         user_fields = user_fields,
                                         place_fields = place_fields,
                                         expansions = expansions,
                                         start_time = start_time,
                                         max_results = max_results)
    return tweets

def get_tweets_df(queries, tweet_fields, user_fields, place_fields, expansions,
    start_time, max_results, name_ex, file_handle=None):

    """
    Retrieves and saves tweet data for a given set of queries.
    The function processes each query, retrieves relevant tweets, and saves the
```

```
    tweet information in an Excel file. It logs each step and the total number of tweets scraped at the end of the process.
    """

    tweet_data = []
    df_tweets = []
    places = {}
    users = {}
    i = 0
    num_rows = 0
    start_position = 0
    total_tweets = 0

    notebook_output = ""

    for query in queries:
        notebook_output += f"{i}\n"

        started_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        notebook_output += f"Processing query: '{query}' | Start Time: {started_time}\n"

        tweets_query = search_tweets(query=query + str(" -is:reply -is:retweet lang:lt"),
                                     tweet_fields=tweet_fields,
                                     user_fields=user_fields,
                                     place_fields=place_fields,
                                     expansions=expansions,
                                     start_time = start_time,
                                     #end_time = end_time,
                                     max_results=max_results)

        if tweets_query.data is not None:
            # Save tweets in a text file
            path = txt_tweet_path + today_date + '_' + query + '.txt'
            with open(path, 'w', encoding='utf-8') as file:
                file.write(str(tweets_query))

            if 'places' in tweets_query.includes:
                places.update({p["id"]: p for p in tweets_query.includes['places']})

            if 'users' in tweets_query.includes:
                users.update({u["id"]: u for u in tweets_query.includes['users']})

            for tweet in tweets_query.data:
                tweet_info = {
                    'Query': query,
                    'Tweet_ID': tweet.id,
                    'Author_ID': tweet.author_id,
```

```
                'Text': tweet.text,
                'Created_At': tweet.created_at,
                'Language': tweet.lang,
                'Public_Metrics': tweet.public_metrics,
                'Attachments': tweet.attachments,
                'Context_Annotations': tweet.context_annotations,
                'Conversation_ID': tweet.conversation_id,
                'Edit_Controls': tweet.edit_controls,
                'Edit_History_Tweet_IDs': tweet.edit_history_tweet_ids,
                'Entities': tweet.entities,
                'In_Reply_To_User_ID': tweet.in_reply_to_user_id,
                'Possibly_Sensitive': tweet.possibly_sensitive,
                'Referenced_Tweets': tweet.referenced_tweets,
                'Reply_Settings': tweet.reply_settings,
                'Source': tweet.source,
                'Withheld': tweet.withheld,
                'Geo': tweet.geo
            }

            if tweet.geo is not None and tweet.geo['place_id'] in places:
                place = places[tweet.geo['place_id']]
                tweet_info['Geo_Name'] = place['full_name']
                tweet_info['Country'] = place['country']
                tweet_info['Short_Geo_Name'] = place['name']
                tweet_info['Place_type'] = place['place_type']

            else:
                tweet_info['Geo_Name'] = ""
                tweet_info['Country'] = ""
                tweet_info['Short_Geo_Name'] = ""
                tweet_info['Place_type'] = ""

            if tweet.author_id is not None and tweet.author_id in users:
                user = users[tweet.author_id]
                tweet_info['Location'] = user['location']
                #tweet_info['User_ID'] = user['id']

            else:
                tweet_info['Location'] = ""
                #tweet_info['User ID'] = ""

            tweet_info['Timestamp'] = started_time
            tweet_data.append(tweet_info)

        df_tweets = pd.DataFrame(tweet_data)

        # Convert datetime column to a specific timezone (e.g., UTC)
          df_tweets['Created_At'] = pd.to_datetime(df_tweets['Created_At'])\
                                      .dt.tz_convert('UTC')
```

```
            # Convert datetime column to string format without timezone information
            df_tweets['Created_At'] = df_tweets['Created_At'].dt\
                                        .strftime('%Y-%m-%d %H:%M:%S')

            if i==0:
                start_position = 0
            else:
                start_position = start_position + num_rows

            # create the excel file if it not already exists
            if not os.path.exists(excel_tweets_path+"Tweets_"+today_date+name_ex+'.xlsx'):
                workbook = xlsxwriter.Workbook(excel_tweets_path+"Tweets_"
                                                  +today_date+name_ex+'.xlsx')
                workbook.close()

            # write data in the excel file with name based on search_phrase_combination
            with pd.ExcelWriter(excel_tweets_path+"Tweets_"+today_date+name_ex+'.xlsx',
                                engine="openpyxl",
                                if_sheet_exists='overlay',
                                mode='a') as writer:
                df_tweets.to_excel(writer,
                                   startrow=start_position,
                                   startcol=0,
                                   index=False,
                                   header=False)

            num_rows = len(df_tweets)
            #print(f"Processing query: '{query}' |     Tweets: {num_rows}")
            notebook_output += f"Processing query: '{query}' | Tweets: {num_rows}\n"
            total_tweets += num_rows

            ended_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            #print(f"Processing query: '{query}' |   End Time: {ended_time}\n")
            notebook_output += f"Processing query: '{query}' | End Time: {ended_time}\n\n"

        else:
            #print("No tweet data available for the query. \n")
            notebook_output += "No tweet data available for the query.\n\n"

        #num_rows = 0
        df_tweets = []
        tweet_data = []
        places = {}
        users = {}
        i = i + 1

```

```
    # Add the total at the end
    notebook_output += f"Total Tweets Scraped: {total_tweets}\n"

    # Print the output to the notebook
    print(notebook_output)

    # If a file handle is provided, save the output to the text file
    if file_handle:
        file_handle.write(notebook_output)
        file_handle.flush()

#--Timelines scraping--------------------------------------------------

class AdapterFix(HTTPAdapter):
    def __init__(self):
        self.ssl_context = ssl.create_default_context()
        super().__init__()

    def init_poolmanager(self, *args, **kwargs):
        kwargs["ssl_context"] = self.ssl_context
        super().init_poolmanager(*args, **kwargs)

    def proxy_manager_for(self, *args, **kwargs):
        kwargs["ssl_context"] = self.ssl_context
        return super().proxy_manager_for(*args, **kwargs)

def Client(bearer_token):
    adapter = AdapterFix()

    client = tweepy.Client(bearer_token=bearer_token)
    client.session.adapters['https://'] = adapter
    # this needs to be done before calling any API methods.
    return client

def get_today():
    # Get today's date
    today = dt.date.today()
    # Format the date as YY_MM_DD
    today_date = today.strftime('%y_%m_%d')
    print(today_date)
    return(today_date)

def get_users_tweets_df(ids, folder, tweet_fields, user_fields, place_fields, expansions, exclude, start_time, end_time, max_results, name_ex, k, bearer_token, file_handle=None):

    today_date = get_today()
    tweet_data = []
    df_tweets = []
```

```
    places = {}
    users = {}
    i = 0
    num_rows = 0
    start_position = 0
    total_tweets = 0

    notebook_output = ""
    notebook_output += f"Beginning of the period | Time: {start_time}\n\n"

    for id in ids:

        #if i == 0:

        notebook_output += f"{id}\n"

        started_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        notebook_output += f"Processing id: '{id}' | Start Time: {started_time}\n"

        tweets_id = get_users_tweets(id=id,
                                     tweet_fields=tweet_fields,
                                     user_fields=user_fields,
                                     place_fields=place_fields,
                                     expansions=expansions,
                                     exclude = exclude,
                                     start_time = start_time,
                                     end_time = end_time,
                                     max_results=max_results,
                                     bearer_token=bearer_token)

        if tweets_id.data is not None:
            # Save tweets in a text file
            path = txt_tweet_path + today_date + '_' + id + '.txt'
            with open(path, 'w', encoding='utf-8') as file:
                file.write(str(tweets_id))

            if 'places' in tweets_id.includes:
                places.update({p["id"]: p for p in tweets_id.includes['places']})

            if 'users' in tweets_id.includes:
                users.update({u["id"]: u for u in tweets_id.includes['users']})

            for tweet in tweets_id.data:
                tweet_info = {
                    'ID': id,
                    'Tweet_ID': tweet.id,
                    'Author_ID': tweet.author_id,
                    'Text': tweet.text,
                    'Created_At': tweet.created_at,
```

```
                'Language': tweet.lang,
                'Public_Metrics': tweet.public_metrics,
                'Attachments': tweet.attachments,
                'Context_Annotations': tweet.context_annotations,
                'Conversation_ID': tweet.conversation_id,
                'Edit_Controls': tweet.edit_controls,
                'Edit_History_Tweet_IDs': tweet.edit_history_tweet_ids,
                'Entities': tweet.entities,
                'In_Reply_To_User_ID': tweet.in_reply_to_user_id,
                'Possibly_Sensitive': tweet.possibly_sensitive,
                'Referenced_Tweets': tweet.referenced_tweets,
                'Reply_Settings': tweet.reply_settings,
                'Source': tweet.source,
                'Withheld': tweet.withheld,
                'Geo': tweet.geo
            }

            if tweet.geo is not None and tweet.geo['place_id'] in places:
                place = places[tweet.geo['place_id']]
                tweet_info['Geo_Name'] = place['full_name']
                tweet_info['Country'] = place['country']
                tweet_info['Short_Geo_Name'] = place['name']
                tweet_info['Place_type'] = place['place_type']

            else:
                tweet_info['Geo_Name'] = ""
                tweet_info['Country'] = ""
                tweet_info['Short_Geo_Name'] = ""
                tweet_info['Place_type'] = ""

            if tweet.author_id is not None and tweet.author_id in users:
                user = users[tweet.author_id]
                tweet_info['Location'] = user['location']
                #tweet_info['User_ID'] = user['id']

            else:
                tweet_info['Location'] = ""
                #tweet_info['User ID'] = ""

            tweet_info['Timestamp'] = started_time

            tweet_data.append(tweet_info)

        df_tweets = pd.DataFrame(tweet_data)

        # Convert datetime column to a specific timezone (e.g., UTC)
        df_tweets['Created_At'] = pd.to_datetime(df_tweets['Created_At'])\
                                      .dt.tz_convert('UTC')

```

```
            # Convert datetime column to string format without timezone information
            df_tweets['Created_At'] = df_tweets['Created_At']\
                                        .dt.strftime('%Y-%m-%d %H:%M:%S')

            if i==0:
                start_position = 0
            else:
                start_position = start_position + num_rows

            # create the excel file if it not already exists
            if not os.path.exists(excel_user_tweets_path
                                    +folder
                                    +"User_tweets_"
                                    +today_date+"_"
                                    +str(k)+name_ex
                                    +'.xlsx'):
                workbook = xlsxwriter.Workbook(excel_user_tweets_path
                +folder
                +"User_tweets_"
                +today_date+"_"
                +str(k)+name_ex+'.xlsx')
                workbook.close()

            # write data in the excel file with name based on search_phrase_combination
            with pd.ExcelWriter(excel_user_tweets_path+folder
                                +"User_tweets_"
                                +today_date
                                +"_"
                                +str(k)
                                +name_ex
                                +'.xlsx',
                                engine="openpyxl",
                                if_sheet_exists='overlay',
                                mode='a') as writer:
                df_tweets.to_excel(writer,
                                   startrow=start_position,
                                   startcol=0,
                                   index=False,
                                   header=False)

            num_rows = len(df_tweets)
            #print(f"Processing id: '{id}' |       Tweets: {num_rows}")
            notebook_output += f"Processing id: '{id}' | Tweets: {num_rows}\n"
            total_tweets += num_rows

            ended_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            #print(f"Processing id: '{id}' |    End Time: {ended_time}\n")
            notebook_output += f"Processing id: '{id}' | End Time: {ended_time}\n\n"

```

```
        else:
            #print("No tweet data available for this user. \n")
            notebook_output += "No tweet data available for this user.\n\n"

        df_tweets = []
        tweet_data = []
        places = {}
        users = {}
        i = i + 1

    #print(f"Enf of the period | Time: {end_time}")
    notebook_output += f"End of the period | Time: {end_time}\n"
    notebook_output += f"Total Tweets Scraped: {total_tweets}\n"  # Add the total at the end

    # Print the output to the notebook
    print(notebook_output)

    # If a file handle is provided, save the output to the text file
    if file_handle:
        file_handle.write(notebook_output)
        file_handle.flush()

    return total_tweets

def call_function_with_breaks(ids, folder, start_time, end_time, bearer_token, kk):
    today_date = get_today()
    chunk_size = 5  # Number of IDs to process in each iteration
    maxi = 100 #tweets for id
    total_tweets_all_iterations = 0

    # Create and open the text file to save the output
    file_path = excel_user_tweets_path+folder
                                       +str(folder)[:-1]
                                       +"_LOG_"
                                       +today_date
                                       +"_"
                                       +str(kk)
                                       +".txt"
    with open(file_path, "w") as file_handle:
        # Iterate over the IDs in chunks of size `chunk_size`
        for i in range(0, len(ids), chunk_size):
            # Get the current chunk of IDs
            chunk = ids[i:i+chunk_size]

            # Call the function with the current chunk of IDs
            total_tweets_iteration = get_users_tweets_df(ids=chunk,
                                                         folder=folder,
                                                         tweet_fields=tweet_fields,
```

```
                                                            user_fields=user_fields,
                                                            place_fields=place_fields,
                                                            expansions=expansions,
                                                            exclude=exclude,
                                                            start_time=start_time,
                                                            end_time=end_time,
                                                            max_results=maxi,
                                                            name_ex = "_"+str(int(i/4)),
                                                            k = kk,
                                                            bearer_token = bearer_token,
                                                            file_handle=file_handle)

            total_tweets_all_iterations += total_tweets_iteration
            print(i+5)
            print(f"Total Tweets Scraped: {total_tweets_all_iterations}")
            time.sleep(15.1 * 60)  # Sleep for 15.1 minutes in seconds
    file_handle.close()  # Close the file after the with block ends

    # Open the text file and write the final total_tweets_all_iterations
    with open(file_path, "a") as file_handle:
        file_handle.write(f"Final Total Tweets Scraped: {total_tweets_all_iterations}\n")

#--- Sentiment analysis ---------------------------------------

flair_sentiment = flair.models.TextClassifier.load('en-sentiment')    #flair
analyzer = SentimentIntensityAnalyzer()                               #vader
classifier = pipeline("sentiment-analysis",
                      model = "finiteautomata/bertweet-base-sentiment-analysis")
afinn = Afinn()

def senti_score(n):
    s = flair.data.Sentence(n)
    flair_sentiment.predict(s)
    total_sentiment = s.labels[0]
    assert total_sentiment.value in ['POSITIVE', 'NEGATIVE']
    sign = 1 if total_sentiment.value == 'POSITIVE' else -1
    score = total_sentiment.score
    return total_sentiment.value[0:3], round(sign * score, 3), sign

def Blob(comment):
    try:
        blob_Lt = TextBlob(str(comment))
        #comment_language = detect(str(comment))
        blob=blob_Lt.translate(from_lang='lt', to='en')
    except:
        blob = "Testing, delete"
    return blob

```

```
def Textblob_fun(blob):
    try:
        if blob.sentiment.subjectivity > 0.3:
            if blob.sentiment.polarity > 0:
                blob_values=1
            else:
                blob_values=-1
        else:
            blob_values=0
    except:
        blob_values=0
    return(blob_values)

def Vader_fun(blob):
    try:
        vs = analyzer.polarity_scores(blob)

        if vs['compound'] >= 0.05:
            vs_value = 1

        elif vs['compound'] < -0.05:
            vs_value = -1
        else:
            vs_value = 0
    except:
        vs_value = 0

    return(vs_value)

def Transformers_fun(blob):
    try:
        transformer_output = classifier(str(blob[0:200]))
        tr_result = transformer_output[0].get('label')
        if tr_result == 'POS':
            tr_value = 1
        elif tr_result == 'NEG':
            tr_value = -1
        else:
            tr_value = 0
    except:
        tr_value = 0

    return(tr_value)

def Flair_fun(blob):
    try:
        total_sentiment, total_score, value  = senti_score(str(blob))
    except:
        value = 0
```

```

    return(value)

def Afinn_fun(blob):
    try:
        afinn_score = afinn.score(str(blob))
        if afinn_score > 0:
            afinn_value = 1
        elif afinn_score < 0:
            afinn_value = -1
        else:
            afinn_value = 0
    except:
        afinn_value = 0

    return(afinn_value)

def sentiments(df_posts, main_folder_dr, excel_name, date):

    #tracking time and fails
    failed = 0
    fail = 0
    total = 0
    start_position = 0
    results_count = 0
    df_with_sentiments = pd.DataFrame()
    posts_ids = []
    post_times = []
    authors = []
    posts = []
    blobs = []
    blob_values = []  # Textblob
    vs_values = []    # Vader
    tr_values = []    # Transformers
    flair_values = [] # Flair
    afinn_values = [] # Afinn

    #Create an excel file
    filepath = main_folder_dr+excel_name+date+'.xlsx'
    wb = openpyxl.Workbook()
    wb.save(filepath)

    for index, row in df_posts.iterrows():

        try:
            #Post_ID
            post_id = row['Tweet_ID']
            posts_ids.append(post_id)

```

```
            #Time
            post_time = row['Created_At']
            post_times.append(str(post_time))

            #Author
            author = row['Author_ID']
            authors.append(str(author))

            #Blob
            comment = row['Text']
            posts.append(comment)

            blob = Blob(comment=comment)
            blobs.append(str(blob))

            #Textblob
            blob_value = Textblob_fun(blob)
            blob_values.append(blob_value)

            #Vader
            vs_value = Vader_fun(blob)
            vs_values.append(vs_value)

            #Transformers
            tr_value = Transformers_fun(blob)
            tr_values.append(tr_value)

            #Flair
            flair_value = Flair_fun(blob)
            flair_values.append(flair_value)

            # Afinn
            afinn_value = Afinn_fun(blob)
            afinn_values.append(afinn_value)

            #Write data to excel
            n = 100
            if (total%n==0 or total == len(df_posts['Text'])-failed-1) and total>0:
                now = datetime.now().strftime("%H:%M:%S")
                print(f'{now} {total-fail}/{total}  {fail}/{n} total failed {failed}')

                if total==n:
                    start_position = 0
                else:
                    start_position = start_position + results_count

                df_with_sentiments=pd.DataFrame({
                        'Tweet_ID': posts_ids,
                        'Post_lt':posts,
```

```
                        'Post_en':blobs,
                        'Created_At': post_times,
                        'Author_ID': authors,
                        'TextBlob values': blob_values,
                        'Vader values': vs_values,
                        'Transformers values': tr_values,
                        'Flair values': flair_values,
                        'Afinn values': afinn_values
                    })

                with pd.ExcelWriter(filepath,
                                    engine="openpyxl",
                                    if_sheet_exists='overlay',
                                    mode='a') as writer:
                    df_with_sentiments.to_excel(writer,
                                                startrow=start_position,
                                                startcol=0,
                                                index=False,
                                                header=False)
                results_count = len(blobs)

                #clear variables
                fail = 0
                df_with_sentiments = pd.DataFrame()

                posts = []
                posts_ids = []
                blobs = []
                post_times = []
                authors = []
                blob_values = []
                vs_values = []    # Vader
                tr_values = []    # Transformers
                flair_values = [] # Flair
                afinn_values = [] # Afinn

            total = total + 1

        except Exception as e:
            print(e)
            #print(total)
            total = total + 1
            failed = failed + 1
            fail = fail+1

#--- Text Preprocessing ---------------------------------------
#Remove twitter handlers
df_md.Text = df_md.Text.apply(lambda x:re.sub('@[^\s]+','',str(x)))
#remove hashtags
```

```python
df_md.Text = df_md.Text.apply(lambda x:re.sub(r'\B#\S+','',x))
# Remove URLS
df_md.Text = df_md.Text.apply(lambda x:re.sub(r"http\S+", "", x))
# Remove all the special characters
df_md.Text = df_md.Text.apply(lambda x:' '.join(re.findall(r'\w+', x)))
#remove all single characters
df_md.Text = df_md.Text.apply(lambda x:re.sub(r'\s+[a-zA-Z]\s+', '', x))
# Remove the '&' character
df_md.Text = df_md.Text.apply(lambda x: re.sub(r'&', '', x))
# Substituting multiple spaces with single space
df_md.Text = df_md.Text.apply(lambda x:re.sub(r'\s+', ' ', x, flags=re.I))
# Replace empty strings and spaces with NaN
df_md['Text'] = df_md['Text'].replace(r'^\s*$', pd.NA, regex=True)
# Drop rows with missing values (NaN) in the "Text" column
df_md.dropna(subset=['Text'], inplace=True)
# Reset the index if needed
df_md.reset_index(drop=True, inplace=True)

#---- SMI calculation ----------------------------------------

# convert 1 0 -1 to POS NEU NEG
def pos_neg_neu(df, index_name):
    sentiments = []
    column_name = index_name.split(' ')[0] + ' result'
    for i in range(0, len(df)):
        if df[index_name][i]>0:
            sentiments.append('POS')
        elif df[index_name][i]<0:
            sentiments.append('NEG')
        else:
            sentiments.append('NEU')
    df[column_name] = sentiments
    return df

def senti_index(df_posts_senti, index_name='TextBlob result', lt=False):

    df_senti = df_posts_senti[['Year', 'Month', index_name]]

    # Count textblob_results
    df = df_senti.groupby(['Year', 'Month', index_name]).size().reset_index(name='Count_left')

    years = df['Year'].unique()
    months = df['Month'].unique()
    textblob_results = df[index_name].unique()

    # Create a MultiIndex DataFrame
    result = pd.DataFrame(index=pd.MultiIndex.from_product([years, months, textblob_results], names=['Year', 'Month', index_name]), columns=['Count_left'])
```

```
    # Rename the column in the joining data
    df_temp = df.set_index(['Year', 'Month', index_name])
    df_temp = df_temp.rename(columns={'Count_left': 'Count'})

    # Join with the original data
    result = result.join(df_temp, how='left')
    # Fill missing values with 0
    result.fillna(0, inplace=True)

    result = result.drop('Count_left', axis=1)
    result = result.sort_values(by=['Year', 'Month'])
    result.reset_index(inplace=True)

    # Calculate sentiment index
    result_neg = result[result[index_name] == 'NEG']
    result_neg.columns = ['Year', 'Month', index_name, 'Count_neg']

    result_pos = result[result[index_name] == 'POS']
    result_pos.columns = ['Year', 'Month', index_name, 'Count_pos']

    result_neu = result[result[index_name] == 'NEU']
    result_neu.columns = ['Year', 'Month', index_name, 'Count_neu']

    # Join pos with neg
    df_pos_neg = pd.merge(result_pos, result_neg, on=['Year', 'Month'], how='inner')
    df_pos_neg = pd.merge(df_pos_neg, result_neu, on=['Year', 'Month'], how='inner')

    if lt == False:
        column_name = index_name.split(' ')[0] + ' SI'
    else:
        column_name = index_name.split(' ')[0] + ' SI' + '_lt'
    df_pos_neg[column_name] = (df_pos_neg['Count_pos'] - df_pos_neg['Count_neg']) /
                              (df_pos_neg['Count_pos'] + df_pos_neg['Count_neg'])
    df_pos_neg["Count"] = (df_pos_neg['Count_pos'] + df_pos_neg['Count_neg'])
    df_pos_neg["Count_all"] = (df_pos_neg['Count_pos']
                              + df_pos_neg['Count_neg']
                              + df_pos_neg['Count_neu'])
    df_pos_neg_full = df_pos_neg
    df_pos_neg["Year_and_Month"] = df_pos_neg['Year'].astype(str)
                                   + '_'
                                   + df_pos_neg['Month'].astype(str)

    # Pad the 'Month' part of 'Year_and_Month'
    # with leading zeros (if needed) to make it two digits
    df_pos_neg['Year_and_Month'] = df_pos_neg['Year_and_Month']\
    .str.split('_')\
    .apply(lambda x: x[0] + '_' + x[1].zfill(2) if len(x[1]) == 1 else x[0] + '_' + x
    [1])
```

```
    df_pos_neg = df_pos_neg[['Year_and_Month', column_name]]
    return pd.DataFrame(df_pos_neg), pd.DataFrame(df_pos_neg_full)

#all
df_TextBlob_senti, df_TextBlob_senti_full = senti_index(df_posts_senti = df_avg,
                                                        index_name='TextBlob result')
df_Vader_senti, df_Vader_senti_full = senti_index(df_posts_senti = df_avg,
                                                  index_name='Vader result')
df_Tr_senti, df_Transformers_senti_full = senti_index(df_posts_senti = df_avg,
                                                      index_name='Transformers result')
df_Flair_senti, df_Flair_senti_full = senti_index(df_posts_senti = df_avg,
                                                  index_name='Flair result')
df_Afinn_senti, df_Afinn_senti_full = senti_index(df_posts_senti = df_avg,
                                                  index_name='Afinn result')

df_senti_list = [df_TextBlob_senti[n:m],
                 df_Vader_senti[n:m],
                 df_Tr_senti[n:m],
                 df_Flair_senti[n:m],
                 df_Afinn_senti[n:m],
                 stat_gov_index[n:m]
                ]
df_SI = reduce(lambda  left,right: pd.merge(left,right,on=['Year_and_Month'],
                                            how='inner'), df_senti_list)


df_senti_list = [
    df_TextBlob_senti_full[["Year_and_Month", "Count"]].rename(columns={"Count": "Textblob_count"}),
    df_Vader_senti_full[["Year_and_Month", "Count"]].rename(columns={"Count": "Vader_count"}),
    df_Transformers_senti_full[["Year_and_Month", "Count"]] .rename(columns={"Count": "Transformers_count"}),
    df_Flair_senti_full[["Year_and_Month", "Count"]].rename(columns={"Count": "Flair_count"}),
    df_Afinn_senti_full[["Year_and_Month", "Count", "Count_all"]].rename(columns={"Count": "Afinn_count"})
                ]
df_count = reduce(lambda  left,right: pd.merge(left,right,on=['Year_and_Month'], how='inner'), df_senti_list)

#---- SARIMAX -----------------------------------------------

def find_stationarity_transformations(variable_series, max_diff=6):
    """
    Find the number of differences required to make a time series stationary.

    variable_series: Pandas Series, the time series to be tested
```

```
    max_diff: int, the maximum number of differencing allowed
    return: A tuple containing the variable name and the number of differences required
    """
    for i in range(max_diff + 1):
        # Apply differencing i times sequentially
        differenced_series = variable_series.copy()
        for _ in range(i):
            differenced_series = differenced_series.diff().dropna()

        # Perform the ADF test
        if not differenced_series.empty:
            result = adfuller(differenced_series, autolag='AIC')
            p_value = result[1]
            if p_value < 0.05:
                # The series is stationary after i differencing
                return (variable_series.name, i)

    # If the series is not stationary after max_diff differencing
    return (variable_series.name, None)

def create_differenced_dataframe(data, stationarity_transformations_df,
    exclude_variable):
    """
    Create a new DataFrame with differenced variables based on the differencing order
    specified in the stationarity_transformations_df. The column names are updated to
    reflect the differencing order.

    data: DataFrame containing the original data
    stationarity_transformations_df: DataFrame containing variables
           and their corresponding differencing order
    exclude_variable: List of variables to exclude from differencing
    return: DataFrame with differenced variables
    """
    data_diff = data.copy()

    for col in data_diff.columns:
        if col not in exclude_variable:
            # Get the number of differencing required for the variable
            diff_order = stationarity_transformations_df[
    stationarity_transformations_df['Variable'] == col]['Differencing_Order'].iloc[0]

            # Apply the differencing sequentially and update column names
            for i in range(diff_order):
                data_diff[col] = data_diff[col].diff()
                # Update the column name only for the last differencing iteration
                if i + 1 == diff_order:
                    new_col_name = f"{col}_diff_{diff_order}"
                    data_diff.rename(columns={col: new_col_name}, inplace=True)

```

```
    # Drop rows with NaN values introduced by differencing
    data_diff = data_diff.dropna()

    return data_diff

# Creating lagged variables for selected lag periods (1, 2, 3, 6, 12 months)
lags = [1, 2, 3, 6, 12]
exog_variables = data_diff_CCI.columns.drop('CCI_diff_1')  # All columns except CCI

# Adding lagged variables to the dataframe
for lag in lags:
    for var in exog_variables:
        data_diff_CCI[f'{var}_lag_{lag}'] = data_diff_CCI[var].shift(lag)

# Dropping initial rows with NaN values due to lags
data_diff_np_lagged = data_diff_CCI.dropna()

# Calculating Pearson and Spearman correlations of lagged variables with CCI
pearson_corr_lagged = data_diff_np_lagged.corr(method='pearson')['CCI_diff_1']
spearman_corr_lagged = data_diff_np_lagged.corr(method='spearman')['CCI_diff_1']

# Filtering the results to only include the lagged variables
pearson_corr_lagged = pearson_corr_lagged.filter(regex='lag')
spearman_corr_lagged = spearman_corr_lagged.filter(regex='lag')

# CCI correlation with differentiated variables

spearman_corr_lagged_series = pd.Series(spearman_corr_lagged)
spearman_corr_corrected_series = pd.Series(spearman_corr_corrected)

# Extract unique variable names (excluding the '_lag_' and digits parts)
unique_vars = set(name.split('_lag_')[0] for name in spearman_corr_lagged.keys())

# Define the order of lags to be shown on the x-axis
lag_order = [0, 1, 2, 3, 6, 12]

# Find the global minimum and maximum correlation values to set a common scale
all_correlations = pd.concat([spearman_corr_lagged_series,
    spearman_corr_corrected_series])
vmin, vmax = all_correlations.min(), 0.35

# Define custom colormap
cmap = LinearSegmentedColormap.from_list("custom_cmap", ["#FFFFFF", "#78003F"])

# Plotting
plt.figure(figsize=(10, 15))

for i, var in enumerate(unique_vars):
    # Filter correlations for the current variable and its lags
```

```
    var_corr_lagged = spearman_corr_lagged_series.filter(regex=f'^{var}_lag_')
    var_corr_unlagged = spearman_corr_corrected_series.filter(regex=f'^{var}(?!.*_lag_)
    ')
    var_corr = pd.concat([var_corr_unlagged, var_corr_lagged])

    # Check if the variable has any correlation data
    if var_corr.empty:
        continue

    # Reshape for plotting
    var_corr_df = pd.DataFrame(var_corr, columns=[var]).T

    # Create heatmap for the current variable
    ax = plt.subplot(len(unique_vars), 1, i+1)
    sns.heatmap(var_corr_df, annot=True, cmap=cmap, center=0, cbar=i == 0,
                xticklabels=lag_order, yticklabels=False,
                                                    vmin=vmin,
                                                    vmax=vmax,
                                                    annot_kws={"size": 20})
    ax.set_title(f'Correlation of CCI with {var}', fontsize=17)
     # Increase x-axis label size
    ax.set_xticklabels(lag_order, fontsize=14)

plt.tight_layout()
plt.show()

#Cointegration
from statsmodels.tsa.vector_ar.vecm import coint_johansen

# Loading the differenced dataset
data_diff = data_diff_CCI

# Selecting the stationary variables after differencing (I(1))
variables_for_cointegration = ['Vader_SI_diff_1',
                               'CCI_diff_1',
                               'Inflation_diff_1',
                               'Unemployment_without_seasonality_diff_1']

# Running the Johansen's cointegration test on the selected variables
coint_test_result = coint_johansen(data_diff[variables_for_cointegration],
                                   det_order=0,
                                   k_ar_diff=1)

# Extracting the test statistics and critical values for interpretation
eigenvalues = coint_test_result.eig
trace_stats = coint_test_result.lr1
max_eigen_stats = coint_test_result.lr2
# Critical values for trace statistic
critical_values_trace = coint_test_result.cvt
```

```
# Critical values for max eigenvalue statistic
critical_values_max_eigen = coint_test_result.cvm

# Prepare results for display
coint_results = {
    "Eigenvalues": eigenvalues,
    "Trace Statistic": trace_stats,
    "Max Eigenvalue Statistic": max_eigen_stats,
    "Critical Values (Trace)": critical_values_trace[:, :3].tolist(),
    "Critical Values (Max Eigen)": critical_values_max_eigen[:, :3].tolist()
}

# Convert the results to a DataFrame
coint_results_df = pd.DataFrame(coint_results)
coint_results_df

#STL
df_senti2['Year_and_Month'] = pd.to_datetime(df_senti2['Year_and_Month'])
df_senti2.set_index('Year_and_Month', inplace=True)

decomposition = STL(df_senti2['CCI'], period=12).fit()
fig, (ax1, ax2, ax3, ax4) = plt.subplots(nrows=4, ncols=1, sharex=True, figsize=(10,8))
ax1.plot(decomposition.observed, color='#E64164')
ax1.set_ylabel('Observed')
ax2.plot(decomposition.trend, color='#E64164')
ax2.set_ylabel('Trend')
ax3.plot(decomposition.seasonal, color='#E64164')
ax3.set_ylabel('Seasonal')
# ax4.plot(decomposition.resid)
# ax4.set_ylabel('Residuals')
ax4.plot(decomposition.resid.index, decomposition.resid, 'o', color='#E64164')
ax4.set_ylabel('Residuals')

# Format the x-axis to show the dates (years)
ax4.xaxis_date()  # Ensure we have a date x-axis
ax4.xaxis.set_major_formatter(plt.matplotlib.dates.DateFormatter("%Y")) # Show only the
    year

# plt.xticks(np.arange(0, 145, 12), np.arange(2018, 2024, 1))
fig.autofmt_xdate()
plt.tight_layout()

def rolling_forecast_sarimax(df, exogs, lags, p, d, q, P, D, Q, s, test_size, plot=
    False):
    """
    Perform a rolling forecast using SARIMAX
    and plot actual vs predicted values with dates on the x-axis.

    df: DataFrame containing the time series
```

```
                and exogenous variables with 'Year_and_Month' as index.
    xog_variables: List of exogenous variables.
    lag_features: Dictionary specifying the lag for each exogenous variable.
    p, d, q: Non-seasonal ARIMA order parameters.
    P, D, Q, s: Seasonal ARIMA order parameters.
    test_size: Number of observations used for testing.
    """
    data = df.copy()
    exog_variables = exogs.copy()
    lag_features = lags.copy()

    # Ensure 'Year_and_Month' is the datetime index
    if 'Year_and_Month' in data.columns:
        data['Year_and_Month'] = pd.to_datetime(data['Year_and_Month'])
        data.set_index('Year_and_Month', inplace=True)

    # Create lagged features
    for feature, lags in lag_features.items():
        for lag in lags:
            data[f'{feature}_lag_{lag}'] = data[feature].shift(lag)

    # Drop rows with NaN values due to lagging
    data = data.dropna()

    # Update exog_variables list to include the lagged features
    for feature, lags in lag_features.items():
        exog_variables += [f'{feature}_lag_{lag}' for lag in lags]

    # Prepare endog and exog data
    endog = data['CCI']
    exog = data[exog_variables]

    # Split the data into training and testing sets
    train_endog = endog.iloc[:-test_size]
    train_exog = exog.iloc[:-test_size]
    test_endog = endog.iloc[-test_size:]
    test_exog = exog.iloc[-test_size:]

    # Lists to store actual and predicted values
    predictions = []

    # Rolling forecast
    for t in range(test_size):
        model = SARIMAX(train_endog,
                        exog=train_exog,
                        order=(p, d, q),
                        seasonal_order=(P, D, Q, s))
        model_fit = model.fit(disp=False)
        next_exog = test_exog.iloc[t:t+1]  # Exog data for the next forecast
```

```
            yhat = model_fit.forecast(exog=next_exog)
            predictions.append(yhat.iloc[0])
            # Update the training set for the next iteration
            train_endog = train_endog.append(test_endog.iloc[t:t+1])
            train_exog = pd.concat([train_exog, next_exog])

        # Calculate error metrics
        mse = mean_squared_error(test_endog, predictions)
        mae = mean_absolute_error(test_endog, predictions)
        rmse_val = sqrt(mse)
        mape_val = np.mean(np.abs((np.array(test_endog) - np.array(predictions)) / np.array
        (test_endog))) * 100

        aic = model_fit.aic

        if plot:
            # Print error metrics
            print(f"MAE: {mae}")
            print(f"MSE: {mse}")
            print(f"RMSE: {rmse_val}")
            print(f"MAPE: {mape_val}%")

            # Plotting the actual values for the entire period and the forecasted values
            plt.figure(figsize=(14, 7))
            plt.plot(endog.index, endog, label='Actual', color='#414141', marker='o')
            forecast_index = endog.index[-test_size:]
            plt.plot(forecast_index,
                     predictions,
                     label='Predicted',
                     color='#E64164',
                     linestyle='--',
                     marker='x')
            plt.axvline(x=forecast_index[0],
                        color='#78003F',
                        linestyle='--',
                        linewidth=2,
                        label='Start of Test Data')  # Separation line
            plt.title('Actual vs Predicted CCI ARIMAX(1,1,1)', fontsize=32)
            plt.legend()
            plt.show()

            # Create results DataFrame with dates
            results_df = pd.DataFrame({
                'Actual': test_endog,
                'Predictions': predictions
            }, index=test_endog.index)
            print(results_df)

        return aic, mae
```

```
def perform_grid_search(data, exog_variables, lag_features, p_range, d, q_range, P_range, D, Q_range, s, test_size):

    """
    Perform a grid search for SARIMAX model parameters.

    data: DataFrame with time series data
    exog_variables: List of exogenous variables
    lag_features: Dictionary with lags for each exogenous variable
    p_range: Range of values for AR parameter
    d: Differencing parameter
    q_range: Range of values for MA parameter
    P_range: Range of values for seasonal AR parameter
    D: Seasonal differencing parameter
    Q_range: Range of values for seasonal MA parameter
    s: Seasonal period
    test_size: Size of the test dataset
    return: DataFrame with grid search results
    """
    results_df = pd.DataFrame(columns=['p', 'd', 'q', 'P', 'D', 'Q', 'AIC', 'MAE'])

    for p in p_range:
        print('-')
        print(p)
        for q in q_range:
            print(q)
            for P in P_range:
                print(P)
                for Q in Q_range:
                    print(Q)
                    exog_vars = exog_variables.copy()
                    aic, mae = rolling_forecast_sarimax(data.copy(),
                                                        exog_vars,
                                                        lag_features,
                                                        p, d, q, P, D, Q, s,
                                                        test_size)
                    results_df = results_df.append({
                        'p': p, 'd': d, 'q': q, 'P': P, 'D': D, 'Q': Q,
                        'AIC': aic,
                        'MAE': mae
                    }, ignore_index=True)

    # Sort the results
    results_df.sort_values(by=['MAE', 'AIC'], inplace=True)
    return results_df

p_range = q_range = P_range = Q_range = range(0, 7)
d = 1
```

```
D = 0
s = 12
test_size = 7
exog_variables = ['Vader_SI_diff_1',
                  'Inflation_diff_1',
                  'Unemployment_without_seasonality_diff_1']
lag_features = {'Unemployment_without_seasonality_diff_1': [6],
               'Inflation_diff_1': [3]}

grid_search_results = perform_grid_search(data_diff_nCCI.copy(), exog_variables,
    lag_features, p_range, d, q_range, P_range, D, Q_range, s, test_size)

# Print the top 5 model combinations
print(grid_search_results.head())

#Forecast with the best parameters
exog_variables = ['Vader_SI_diff_1',
                  'Inflation_diff_1',
                  'Unemployment_without_seasonality_diff_1']
lag_features = {'Unemployment_without_seasonality_diff_1': [6],
               'Inflation_diff_1': [3]}

p, d, q, P, D, Q, s = 3, 1, 5, 0, 0, 3, 12  # Example SARIMA order parameters

test_size = 7  # Example test size
rolling_forecast_sarimax(data_diff_nCCI.copy(),
                         exog_variables,
                         lag_features,
                         p, d, q, P, D, Q, s,
                         test_size, plot = True)

#Decomposition
def rolling_forecast_sarimax_with_coefficients(df, exogs, lags, p, d, q, P, D, Q, s,
    test_size, plot=False):
    """
    Perform a rolling forecast using SARIMAX, saving the model coefficients at each
    step.
    """
    data = df.copy()
    exog_variables = exogs.copy()
    lag_features = lags.copy()

    # Ensure 'Year_and_Month' is the datetime index
    if 'Year_and_Month' in data.columns:
        data['Year_and_Month'] = pd.to_datetime(data['Year_and_Month'])
        data.set_index('Year_and_Month', inplace=True)

    # Create lagged features
    for feature, lags in lag_features.items():
```

```
        for lag in lags:
            data[f'{feature}_lag_{lag}'] = data[feature].shift(lag)

    # Drop rows with NaN values due to lagging
    data = data.dropna()

    # Update exog_variables list to include the lagged features
    for feature, lags in lag_features.items():
        exog_variables += [f'{feature}_lag_{lag}' for lag in lags]

    # Prepare endog and exog data
    endog = data['CCI']
    exog = data[exog_variables]

    # Split the data into training and testing sets
    train_endog = endog.iloc[:-test_size]
    train_exog = exog.iloc[:-test_size]
    test_endog = endog.iloc[-test_size:]
    test_exog = exog.iloc[-test_size:]

    # Lists to store actual and predicted values, and coefficients
    predictions = []
    coefficients = []

    # Rolling forecast
    for t in range(test_size):
        model = SARIMAX(train_endog,
                        exog=train_exog,
                        order=(p, d, q),
                        seasonal_order=(P, D, Q, s))
        model_fit = model.fit(disp=False)
        next_exog = test_exog.iloc[t:t+1]  # Exog data for the next forecast
        yhat = model_fit.forecast(exog=next_exog)
        predictions.append(yhat.iloc[0])
        coefficients.append(model_fit.params)
        print(model_fit.summary())
        model_fit.plot_diagnostics(figsize=(10,8));

       # Assuming model_fit is your fitted SARIMAX model
        residuals = model_fit.resid

        plt.tight_layout()
        plt.show()

        # Update the training set for the next iteration
        train_endog = train_endog.append(test_endog.iloc[t:t+1])
        train_exog = pd.concat([train_exog, next_exog])

    # Calculate error metrics
```

```python
    mse = mean_squared_error(test_endog, predictions)
    mae = mean_absolute_error(test_endog, predictions)
    rmse_val = sqrt(mse)
    mape_val = np.mean(np.abs((np.array(test_endog) - np.array(predictions))
                             / np.array(test_endog))) * 100
    aic = model_fit.aic

    # Create DataFrame of coefficients
    coefficients_df = pd.DataFrame(coefficients, index=test_endog.index)

    if plot:
        # Plotting actual vs predicted values
        plt.figure(figsize=(12, 6))
        plt.plot(endog.index, endog, label='Actual', color='blue', marker='o')
        forecast_index = endog.index[-test_size:]
        plt.plot(forecast_index,
                 predictions,
                 label='Predicted',
                 color='red',
                 linestyle='--',
                 marker='x')
        plt.axvline(x=forecast_index[0],
                    color='grey',
                    linestyle='--',
                    linewidth=2,
                    label='Start of Test Data')
        plt.title('Actual vs Predicted CCI with SARIMAX')
        plt.legend()
        plt.show()

        # Print error metrics and results DataFrame
        print(f"MAE: {mae}, MSE: {mse}, RMSE: {rmse_val}, MAPE: {mape_val}%")
        results_df = pd.DataFrame({'Actual': test_endog,
                                   'Predictions': predictions},
                                  index=test_endog.index)
        print(results_df)

    return aic, mae, coefficients_df

p, d, q, P, D, Q, s = 3, 1, 5, 0, 0, 3, 12  # Example ARIMA order parameters

# Example exogenous variables and lags (adjust based on your data)
exog_variables = ['Vader_SI_diff_1',
                  'Inflation_diff_1',
                  'Unemployment_without_seasonality_diff_1']
lag_features = {'Unemployment_without_seasonality_diff_1': [6]}

# Call the function
aic, mae, coefficients_df = rolling_forecast_sarimax_with_coefficients(
```

```
    df=data_diff_nCCI,    # Replace with your DataFrame name
    exogs=exog_variables,
    lags=lag_features,
    p=p, d=d, q=q,
    P=P, D=D, Q=Q,
    s=s,
    test_size=test_size,
    plot=True  # Set to True if you want to plot the results
)

#—— VECM ——————————————————————————————————————

def rolling_forecast_vecm(df, original_cci_data, exog_variables, max_lag, test_size,
    plot=False):
    """
    Perform a rolling forecast using VECM and plot actual vs predicted values.

    df: DataFrame containing the time series and exogenous variables.
    original_cci_data: DataFrame containing the original (undifferenced) CCI data.
    exog_variables: List of exogenous variables.
    max_lag: Maximum lag order for the VECM model.
    test_size: Number of observations used for testing.
    plot: Boolean flag to plot the actual vs predicted values.
    """
    data = df.copy()
    original_cci = original_cci_data.copy()

    # Ensure 'Year_and_Month' is the datetime index
    if 'Year_and_Month' in data.columns:
        data['Year_and_Month'] = pd.to_datetime(data['Year_and_Month'])
        data.set_index('Year_and_Month', inplace=True)
    if 'Year_and_Month' in original_cci.columns:
        original_cci['Year_and_Month'] = pd.to_datetime(original_cci['Year_and_Month'])
        original_cci.set_index('Year_and_Month', inplace=True)

    # Preparing the VECM data with the specified exogenous variables and target
    variable
    vecm_data = data[exog_variables + ['CCI_diff_1']]

    # Rolling forecast
    predictions = []
    for t in range(test_size):
        # Training data excludes the test set
        train_data = vecm_data.iloc[:-test_size+t]
        # Fitting the VECM model
        vecm_model = VECM(train_data, k_ar_diff=max_lag, coint_rank=1)
        vecm_fit = vecm_model.fit()
        # Forecasting the next step
        forecast = vecm_fit.predict(steps=1)
```

```
        predictions.append(forecast[0, -1])  # Assuming CCI_diff_1 is the last column

    # Calculate error metrics for integrated forecasts
    # Last actual undifferenced CCI value
    last_actual_undiff_cci = original_cci['CCI'].iloc[-test_size-1]
    integrated_forecasts = last_actual_undiff_cci + np.cumsum(predictions)
    mae_integrated = mean_absolute_error(original_cci['CCI'].iloc[-test_size:],
                                         integrated_forecasts)

    actual = data['CCI_diff_1'].iloc[-test_size:]
    mse = mean_squared_error(original_cci['CCI'].iloc[-test_size:],
    integrated_forecasts)
    mae = mean_absolute_error(original_cci['CCI'].iloc[-test_size:],
    integrated_forecasts)
    #mae = mean_absolute_error(actual, integrated_forecasts)
    rmse_val = sqrt(mse)

    if plot:
        # Integrating the differenced forecasts to obtain the level forecasts for CCI
        # Last actual undifferenced CCI value
        last_actual_undiff_cci = original_cci['CCI'].iloc[-test_size-1]
        integrated_forecasts = last_actual_undiff_cci + np.cumsum(predictions)

        # Plotting the forecast results for the entire period
        plt.figure(figsize=(14, 7))
        plt.plot(original_cci['CCI'][-len(original_cci)+test_size:],
                 label='Actual', color='#414141', marker='o')
        plt.plot(original_cci.index[-test_size:],
                                  integrated_forecasts,
                                  label='Predicted',
                                  color='#E64164',
                                  linestyle='--',
                                  marker='x')
        plt.axvline(x=original_cci.index[-test_size],
                                      color='#78003F',
                                      linestyle='--',
                                      linewidth=2,
                                      label='Start of Test Data')
        plt.title('Actual vs Forecasted CCI (VECM)', fontsize=20)
        plt.xlabel('Time Periods')
        plt.ylabel('CCI')
        plt.legend(loc='lower left')
        plt.show()

    return mae, mse, rmse_val

# the first try
exog_variables = ['Vader_SI_diff_1', 'Inflation_diff_1',
                  'Unemployment_without_seasonality_diff_1']
```

```
max_lag = 6
test_size = 7
mae, mse, rmse = rolling_forecast_vecm(data_diff, original_cci_data,
                                       exog_variables, max_lag, test_size, plot=True)
print(f"MAE: {mae}, MSE: {mse}, RMSE: {rmse}") #mae 0.14998

# parameters combination search
def test_exog_lag_combinations(df, original_cci_data, exog_vars, max_lags, test_size):
    results = []
    # Generate all non-empty combinations of exogenous variables
    for i in range(1, len(exog_vars) + 1):
        for combo in combinations(exog_vars, i):
            # Test each combination with all specified lag lengths
            for lag in range(1, max_lags + 1):
                # Convert the combo to a list
                # so it can be passed to the forecasting function
                exog_combo = list(combo)
                mae, mse, rmse = rolling_forecast_vecm(
                    df, original_cci_data, exog_combo, lag, test_size, plot=False
                )
                # Store the results
                results.append({
                    'combo': ''.join([var[0] for var in exog_combo]),
                    'mae': mae,
                    'max_lags': lag
                })

                print(results[-1])

    # Convert the results to a DataFrame and sort by MAE
    results_df = pd.DataFrame(results)
    results_df_sorted = results_df.sort_values(by='mae').reset_index(drop=True)
    return results_df_sorted

# Define the exogenous variables and parameters
exog_variables = ['Vader_SI_diff_1',
                  'Inflation_diff_1',
                  'Unemployment_without_seasonality_diff_1']
max_lags = 12

# Call the function with the datasets and parameters
sorted_results = test_exog_lag_combinations(
    df=data_diff,
    original_cci_data=original_cci_data,
    exog_vars=exog_variables,
    max_lags=max_lags,
    test_size=test_size
)

```

```
# the second try
exog_variables = ['Unemployment_without_seasonality_diff_1']
max_lag = 4
mae, mse, rmse = rolling_forecast_vecm(data_diff,
                                       original_cci_data,
                                       exog_variables,
                                       max_lag,
                                       test_size,
                                       plot=True)
print(f"MAE: {mae}, MSE: {mse}, RMSE: {rmse}") #MAE: 0.01596954

#-- Random Forest ----------------------------------------------

def plot_predictions(df, y_pred, target_column,
                     test_size, prediction_label='Predicted CCI'):
    """
    Plots the actual vs predicted values. Assumes the DataFrame index is a DateTimeIndex.
    Parameters:
    df (DataFrame): The original DataFrame containing the target column.
    y_pred (array): The predicted values.
    target_column (str): The name of the target column in df.
    prediction_label (str): Label for the predicted values in the plot.
    """
    df2 = df.copy()

    # Ensure 'Year_and_Month' is in a DateTime format and set as index
    if 'Year_and_Month' in df2.columns:
        df2['Year_and_Month'] = pd.to_datetime(df2['Year_and_Month'])
        df2.set_index('Year_and_Month', inplace=True)

    # Determine the start date of the test data within the df
    actual_start_of_test = df2.index[-test_size]

    # Assuming the last entries in df correspond to the length of y_pred
    dates_for_pred = df2.index[-len(y_pred):]  # This should align with the test set

    plt.figure(figsize=(14, 7))
    plt.plot(df2.index, df2[target_column],
             label='Actual ' + target_column,
             color='#414141', marker='o')
    plt.plot(dates_for_pred, y_pred,
             label=prediction_label,
             color='#E64164', linestyle='--', marker='x')
    plt.title('Actual vs Predicted CCI Random Forest', fontsize=16)
    plt.axvline(x=actual_start_of_test, color='#78003F',
                linestyle='--', linewidth=2, label='Start of Test Data')
    plt.xlabel('Date')
    plt.ylabel(target_column)
```

```python
    plt.legend(loc='lower left')
    plt.show()

def plot_feature_importances(model, feature_names, importance_threshold=0.01):
    """
    Plots the feature importances of a given model,
    filtering out features below a certain threshold.
    Parameters:
    model: The trained model.
    feature_names (list): List of feature names.
    importance_threshold (float):
    The threshold for feature importance (default is 0.01, or 1%).
    """
    importances = model.feature_importances_
    # Filter out features with importance below the threshold
    filtered_indices = [i for i, imp in enumerate(importance)
                        if imp > importance_threshold]
    filtered_importances = [importances[i] for i in filtered_indices]
    filtered_feature_names = [feature_names[i] for i in filtered_indices]
    indices = np.argsort(filtered_importances)[::-1]
    plt.figure(figsize=(10, 4))
    plt.title("Feature Importances")
    plt.bar(range(len(filtered_feature_names)),
            np.array(filtered_importances)[indices],
            color="r", align="center")
    plt.xticks(range(len(filtered_feature_names)),
               [filtered_feature_names[i] for i in indices], rotation=90)
    plt.xlim([-1, len(filtered_feature_names)])
    plt.show()

def summarize_model_errors(model_errors, decimals=5):
    """
    Summarizes the errors (MAE, MSE, RMSE, MAPE)
    of a model into a DataFrame, with rounded values.
    Parameters:
    model_errors (dict): A dictionary containing
    MAE, MSE, RMSE, and MAPE for the model.
    decimals (int): Number of decimal places to round to.
    Returns:
    pd.DataFrame: A DataFrame summarizing the errors,
    with each error as a row.
    """
    error_df = pd.DataFrame(model_errors.items(),
                            columns=['Error Metric', 'Value'])
    error_df['Value'] = error_df['Value'].round(decimals)
    return error_df

def summarize_feature_importances(model, feature_names, importance_threshold=0.01):
    """
```

```python
    Creates a DataFrame of the feature importances in a model,
    filtering out those below a certain threshold.
    Parameters:
    model: The trained model.
    feature_names (list): List of feature names.
    importance_threshold (float):
    The threshold for feature importance (default is 0.01, or 1%).
    Returns:
    pd.DataFrame: A DataFrame listing significant feature importances.
    """
    importances = model.feature_importances_
    features_df = pd.DataFrame({'Feature': feature_names, 'Importance': importances})
    features_df = features_df[features_df['Importance'] > importance_threshold]
    features_df.sort_values(by='Importance', ascending=False, inplace=True)
    features_df['Rank'] = range(1, len(features_df) + 1)
    features_df.set_index('Rank', inplace=True)
    features_df['Feature'] = features_df['Feature']\
    + ' (' + round(features_df['Importance'], 4).astype(str) + ')'
    return features_df[['Feature']]

def display_formatted_tables(error_summary, feature_importance_summary):
    """
    Displays the error summary and feature importance tables in a formatted manner.
    Parameters:
    error_summary (pd.DataFrame): DataFrame containing error metrics.
    feature_importance_summary (pd.DataFrame): DataFrame containing feature importances
    .
    """
    # Set display options for better console output
    pd.set_option('display.max_columns', None)
    pd.set_option('display.expand_frame_repr', False)
    pd.set_option('display.max_colwidth', None)
    pd.set_option('display.precision', 4)

    print("\nFeature Importance Summary:")
    print(feature_importance_summary)

    print("\nModel Error Summary:")
    print(error_summary)

    # Reset display options to default
    pd.reset_option('display.max_columns')
    pd.reset_option('display.expand_frame_repr')
    pd.reset_option('display.max_colwidth')
    pd.reset_option('display.precision')

def display_formatted_feature_importances(feature_importances_summaries):
    """
    Displays feature importance tables
```

```
    for each step in a formatted manner.
    Parameters:
    feature_importances_summaries (list):
    List of DataFrames, each containing feature importances for a step.
    """
    # Set display options for better console output
    pd.set_option('display.max_columns', None)
    pd.set_option('display.expand_frame_repr', False)
    pd.set_option('display.max_colwidth', None)
    pd.set_option('display.precision', 4)

    for i, summary in enumerate(feature_importances_summaries, 1):
        print(f"\nFeature Importance Summary at Step {i}:")
        print(summary)

    # Reset display options to default
    pd.reset_option('display.max_columns')
    pd.reset_option('display.expand_frame_repr')
    pd.reset_option('display.max_colwidth')
    pd.reset_option('display.precision')

def collect_feature_importances(feature_importances_summaries, feature_names):
    """
    Collects feature importances from summaries
    and stores them in a format suitable for creating a DataFrame.

    Parameters:
    feature_importances_summaries (list):
    List of DataFrames, each containing feature importances for a step.
    feature_names (list): List of feature names.

    Returns:
    dict: A dictionary containing feature importances for each step,
    organized by feature name.
    """
    feature_importances_collected = {feature: [] for feature in feature_names}

    for summary in feature_importances_summaries:
        for feature in feature_names:
            if feature in summary.index:
                feature_importances_collected[feature].append(summary.loc[feature,
                                                                          'Importance'])
            else:
                feature_importances_collected[feature].append(0)

    return feature_importances_collected

def combine_feature_importances(feature_importances_collected):
```

```
    """
    Combines collected feature importances into a DataFrame.

    Parameters:
    feature_importances_collected (dict): A dictionary
    containing feature importances for each step.

    Returns:
    pd.DataFrame: A DataFrame combining feature importances across steps.
    """
    return pd.DataFrame(feature_importances_collected)

def tune_hyperparameters(X_train, y_train):
    # Define the parameter grid
    param_grid = {
        'n_estimators': [100, 200, 300],
        'max_depth': [10, 20, 30, None],
        'min_samples_split': [2, 5],
        'min_samples_leaf': [1, 2],
        'max_features': ['auto', 'sqrt']
    }

    # Initialize the Random Forest model
    rf = RandomForestRegressor(random_state=42)

    # Perform Randomized Search
    rf_random = RandomizedSearchCV(estimator=rf,
                                   param_distributions=param_grid,
                                   n_iter=10, cv=3,
                                   random_state=42, n_jobs=-1)
    rf_random.fit(X_train, y_train)

    # Return the best parameters or the best estimator
    return rf_random.best_params_

def rolling_forecast_with_lags(df, lag_features,
                               n_of_lags, rolling_window,
                               test_size,
                               show_plots=True, show_tables=True):
    """
    Perform a rolling forecast with lagged features
    and rolling averages using Random Forest.

    Parameters:
    df (pd.DataFrame): The dataset to use.
    lag_features (list): List of column names
    to create lagged and rolling average features.
    n_of_lags (int): Number of lag periods.
    rolling_window (int): Window size for rolling averages.
```

```
    train_ratio (float): Ratio of data to be used for training.

    Returns:
    dict: A dictionary containing predictions, actuals, MAE, and RMSE.
    """
    data = df[lag_features].copy()

    # Convert 'Year_and_Month' to datetime and set it as the index (if applicable)
    if 'Year_and_Month' in data.columns:
        data['Year_and_Month'] = pd.to_datetime(data['Year_and_Month'])
        data.set_index('Year_and_Month', inplace=True)

    # Create lagged and rolling average features
    for feature in lag_features:
        for lag in range(1, n_of_lags + 1):
            data[f'{feature}_lag_{lag}'] = data[feature].shift(lag)
            data[f'{feature}_rolling_avg_{rolling_window}'] \
            = data[feature].rolling(window=rolling_window).mean().shift(1)

    # Remove NaNs created by lagging and rolling
    data = data.dropna()
    split_index = len(data) - test_size
    train, test = data.iloc[:split_index], data.iloc[split_index:]

    # Get the best hyperparameters
    best_params = tune_hyperparameters(train.drop('CCI', axis=1), train['CCI'])
    print(best_params)

    # Initialize Random Forest Regressor with best hyperparameters
    rf_model = RandomForestRegressor(**best_params, random_state=42)

    # Train the model
    X_train = train.drop('CCI', axis=1)
    y_train = train['CCI']
    rf_model.fit(X_train, y_train)

    # Rolling forecast
    predictions = []
    feature_importances_summaries = []

    for i in range(len(test)):
        # Calculate and print rolling averages
        #for each variable before the prediction
        if show_tables:
            for feature in lag_features:
                rolling_avg = train[feature]\
                .rolling(window=rolling_window).mean().iloc[-1]

        X_test = test.iloc[i:i+1].drop('CCI', axis=1)
```

```
        prediction = rf_model.predict(X_test)[0]
        predictions.append(prediction)

        # Add the actual value to the training set
        # and re-train the model
        new_row = test.iloc[i]
        train = train.append(new_row)
        X_train = train.drop('CCI', axis=1)
        y_train = train['CCI']
        rf_model.fit(X_train, y_train)

        # Collect feature importances
        feature_importance_summary = summarize_feature_importances(rf_model,
                                                                   X_train.columns)
        feature_importances_summaries.append(feature_importance_summary)

    # Calculate MAE, MSE, RMSE
    mae = mean_absolute_error(test['CCI'], predictions)
    mse = mean_squared_error(test['CCI'], predictions)
    rmse = np.sqrt(mse)

    # Calculate errors
    model_errors = {
        'MAE': mae,
        'MSE': mse,
        'RMSE': rmse
    }

    if show_plots:
        # Diagnostics printout
        print("Diagnostics:")
        print("Length of predictions array:", len(predictions))
        print("Length of test set:", len(test))
        print("Start date for predictions:", test.index[0])
        print("Start date for test set:", df.index[split_index])
        print("First few predicted values:", predictions[:3])
        print("First few actual test values:", test['CCI'].head(3).tolist())

        # Plot predictions
        plot_predictions(df=df,
                         y_pred=predictions,
                         target_column='CCI',
                         test_size=test_size,
                         prediction_label='Predicted CCI')

    # Create and display error summary
    error_summary = summarize_model_errors(model_errors)
    print("\nModel Error Summary:")
    print(error_summary)
```

```

    if show_tables:

        # Display feature importance summaries for each step
        display_formatted_feature_importances(feature_importances_summaries)

    results_df = pd.DataFrame()
    results_df['Actual'] = df['CCI'][-len(predictions):]
    results_df['Predictions'] = predictions

    print(f'\n{results_df}')
    return {
        'predictions': predictions,
        #'actuals': actuals,
        'mae': mae,
        'mse': mse,
        'rmse': rmse,
    }

def rolling_forecast_with_custom_lags2(data, custom_lags, rolling_variables, test_size,
     show_plots=True, show_tables=True):
    """
    Perform a rolling forecast with custom lagged features
    and rolling average features using Random Forest.
    Returns:
    dict: A dictionary containing predictions, actuals, MAE, MSE, and RMSE.
    """

    df = data.copy()

    # Ensure 'Year_and_Month' is in datetime format and set as index
    if 'Year_and_Month' in df.columns:
        df['Year_and_Month'] = pd.to_datetime(df['Year_and_Month'])
        df.set_index('Year_and_Month', inplace=True)

    # Create custom lagged features
    for feature, lags in custom_lags.items():
        for lag in lags:
            if lag == 0:
                # Include the current period's value if lag is 0
                df[f'{feature}_lag_{lag}'] = df[feature]
            else:
                df[f'{feature}_lag_{lag}'] = df[feature].shift(lag)

    # Create rolling average features
    for feature, window in rolling_variables.items():
        df[f'{feature}_rolling_avg_{window}'] \
        = df[feature].rolling(window=window).mean().shift(1)

```

```
    # Combine selected features
    lag_features = [f"{feature}_lag_{lag}" for feature, lags in custom_lags.items() for
     lag in lags]
    rolling_features = [f"{feature}_rolling_avg_{window}" for feature, window in
    rolling_variables.items()]
    selected_features = lag_features + rolling_features

    # Keep only the selected features and the target variable 'CCI'
    data = df[['CCI'] + selected_features].dropna()

    # Split dataset into training and testing sets
    split_index = len(data) - test_size
    train, test = data.iloc[:split_index], data.iloc[split_index:]

    # Prepare training data
    X_train = train.drop('CCI', axis=1)
    y_train = train['CCI']

    # Get the best hyperparameters
    best_params = tune_hyperparameters(X_train, y_train)

    # Initialize and fit the Random Forest Regressor
    rf_model = RandomForestRegressor(**best_params, random_state=42)
    rf_model.fit(X_train, y_train)

    # Rolling forecast
    predictions = []
    feature_importances_summaries = []
    for i in range(len(test)):
        X_test = test.iloc[i:i+1].drop('CCI', axis=1)
        prediction = rf_model.predict(X_test)[0]
        predictions.append(prediction)

        # Add actual value to the training set and re-train
        new_row = test.iloc[i]
        train = train.append(new_row)
        X_train = train.drop('CCI', axis=1)
        y_train = train['CCI']
        rf_model.fit(X_train, y_train)

        # Collect feature importance
        feature_importance_summary = summarize_feature_importances(rf_model, X_train.
    columns)
        feature_importances_summaries.append(feature_importance_summary)

    # Calculate errors
    model_errors = {
        'MAE': mean_absolute_error(test['CCI'], predictions),
        'MSE': mean_squared_error(test['CCI'], predictions),
```

```
        'RMSE': np.sqrt(mean_squared_error(test['CCI'], predictions))
    }

    # Plot predictions and display tables if requested
    if show_plots:
        plot_predictions(df, predictions, 'CCI', test_size, 'Predicted CCI')
        error_summary = summarize_model_errors(model_errors)
        print("\nModel Error Summary:")
        print(error_summary)
    if show_tables:
        display_formatted_feature_importances(feature_importances_summaries)

    results_df = pd.DataFrame()
    results_df['Actual'] = df['CCI'][-len(predictions):]
    results_df['Predictions'] = predictions

    return {
        'predictions': predictions,
        'model_errors': model_errors,
        'results_df': results_df
    }

def create_feature_importance_subplot(data_steps):
    """
    Creates a 4x2 subplot of horizontal bar charts to display feature importance for each step.
    Parameters:
    - data_steps (dict): A dictionary with steps as keys and lists of (feature, importance) tuples as values.
    """
    # Creating the subplot
    fig, axes = plt.subplots(4, 2, figsize=(15, 20))
    axes = axes.flatten()  # Flatten the axes array for easy iteration

    # Plotting data for each step
    for i, (step, values) in enumerate(data_steps.items()):
        ax = axes[i]
        df = pd.DataFrame(values, columns=["Feature", "Importance"])
        bars = ax.barh(df["Feature"], df["Importance"], color='#78003F')
        ax.set_title(f'{step}')
        ax.set_xlabel('Importance')
        ax.invert_yaxis()

        # Add text labels
        for index, bar in enumerate(bars):
            # Place the text inside the bar for the first bar, otherwise outside
            if index == 0:  # First bar
                text_x_pos = bar.get_width() / 1.15  # 2 Center of the bar
                text_color = 'white'
```

```
            else:
                text_x_pos = bar.get_width() + 0.005  # Slightly outside the bar
                text_color = 'black'

            ax.text(text_x_pos, bar.get_y() + bar.get_height() / 2,
                    f'{bar.get_width():.4f}', va='center', color=text_color, fontsize=8)

    # Hide the empty subplot if the number of steps is odd
    if len(data_steps) % 2 != 0:
        axes[-1].axis('off')

    # Adjust layout
    plt.tight_layout()
    plt.show()

#call RF with  all features and their lags up to 12
custom_lags = {
    'TextBlob_SI': list(range(0, 13)),
    'Vader_SI': list(range(0, 13)),
    'Transformers_SI': list(range(0, 13)),
    'Flair_SI': list(range(0, 13)),
    'Afinn_SI': list(range(0, 13)),
    'CCI': list(range(1, 13)),  # Assuming CCI is the target, start from lag 1
    'Average_wage': list(range(0, 13)),
    'Pension': list(range(0, 13)),
    'Inequality': list(range(0, 13)),
    'Inflation': list(range(0, 13)),
    'Unemployment_without_seasonality': list(range(0, 13)),
    'Unemployment_rate': list(range(0, 13))
}

rolling_variables = {}

results = rolling_forecast_with_custom_lags2(data=df_senti2,
                                             custom_lags=custom_lags,
                                             rolling_variables=rolling_variables,
                                             test_size=7,
                                             show_plots=True,
                                             show_tables=True)

#call RF with  selected features and their lags
custom_lags = {
    'CCI': [1],
    'TextBlob_SI': [0, 1, 7],
    'Inflation': [3],
    'Pension': [3],
}

```

```
rolling_variables = {}

results = rolling_forecast_with_custom_lags2(data=df_senti2, custom_lags=custom_lags,
    rolling_variables=rolling_variables, test_size=7, show_plots=True, show_tables=True)

#call RF with  selected features, their lags and rolling averages
rolling_variables = {
    'Inflation': 4,
    'Pension': 5,
    'CCI': 4,

}

results = rolling_forecast_with_custom_lags2(data=df_senti2, custom_lags=custom_lags,
    rolling_variables=rolling_variables, test_size=7, show_plots=True, show_tables=True)

#-- XGBoost ----------------------------------------

def rolling_forecast_xgboost(data, target_column, date_column, lag_features, test_size,
     rolling_variables, plot=True):
    """
    Performs rolling forecasting with an XGBoost model.

    Parameters:
    - data: DataFrame containing the dataset.
    - target_column: The name of the target variable column.
    - date_column: The name of the date column.
    - lag_features: Dictionary of features with their respective lags (including 0 for
    current values).
    - test_size: Number of last values to forecast.
    - rolling_variables: Dictionary of features with their respective rolling window
    sizes.

    Returns:
    - pd.DataFrame: A dataframe with actual and predicted values.
    """
    df = data.copy()
    df_sorted = df.sort_values(by=date_column).reset_index(drop=True)

    # Prepare the dataset with lagged and rolling features
    for feature, lags in lag_features.items():
        for lag in lags:
            column_name = f'{feature}_lag_{lag}'
            df_sorted[column_name] = df_sorted[feature].shift(lag)

    for feature, window in rolling_variables.items():
        column_name = f'{feature}_rolling_avg_{window}'
        df_sorted[column_name] = df_sorted[feature].rolling(window=window).mean().shift
    (1)
```

```
    # Drop rows with NaN values
    df_sorted.dropna(inplace=True)

    # Define features for the model
    model_features = [f'{feature}_lag_{lag}' for feature in lag_features for lag in lag_features[feature]] + \
                     [f'{feature}_rolling_avg_{rolling_variables[feature]}' for feature in rolling_variables]

    # Splitting the data into training and testing sets
    train_size = len(df_sorted) - test_size
    results_df = pd.DataFrame(columns=[date_column, 'Actual', 'Predictions'])

    for step in range(test_size):
        X_train = df_sorted.iloc[:train_size + step][model_features]
        y_train = df_sorted.iloc[:train_size + step][target_column]

        # Train the XGBoost model
        model = xgb.XGBRegressor(objective='reg:squarederror')
        model.fit(X_train, y_train)

        # Plot feature importance after each step
        #plot_feature_importance(model, model_features, step, target_column)
        if plot:
            # Summarize feature importance after each step
            summarize_feature_importance(model, model_features, step)

        # Prepare the next test point
        next_test_point = df_sorted.iloc[[train_size + step]][model_features]
        prediction = model.predict(next_test_point)[0]

        # Append the prediction and actual value to the results dataframe
        results_df = results_df.append({
            date_column: df_sorted.iloc[train_size + step][date_column],
            'Actual': df_sorted.iloc[train_size + step][target_column],
            'Predictions': prediction
        }, ignore_index=True)

    if plot:
        # Calculate and print metrics
        mae = mean_absolute_error(results_df['Actual'], results_df['Predictions'])
        mse = mean_squared_error(results_df['Actual'], results_df['Predictions'])
        rmse = np.sqrt(mse)
        print(f"\nMean Absolute Error: {mae}")
        print(f"Mean Squared Error: {mse}")
        print(f"Root Mean Squared Error: {rmse}")

        # Plotting actual vs predicted values and feature importance
```

```python
        plot_results_and_feature_importance(df_sorted, results_df, model, date_column, target_column, train_size, model_features)

    return results_df

def plot_results_and_feature_importance(df_sorted, results_df, model, date_column, target_column, train_size, model_features):
    """
    Plots the actual vs predicted values and feature importance.
    """
    plt.figure(figsize=(12, 6))
    plt.plot(df_sorted[date_column], df_sorted[target_column], label='Actual', color='#414141', marker='o')
    plt.plot(results_df[date_column], results_df['Predictions'], label='Predicted', color='#E64164', linestyle='dashed', marker='x')
    plt.axvline(x=df_sorted[date_column].iloc[train_size], color='#78003F', linestyle='--', label='Start of Test Data')
    plt.xlabel(date_column)
    plt.ylabel(target_column)
    plt.title(f'Actual vs Predicted {target_column} - XGBoost Rolling Forecast')
    plt.legend()
    plt.show()

    feature_importance = model.feature_importances_
    feature_importance_df = pd.DataFrame({'Feature': model_features, 'Importance': feature_importance})
    feature_importance_df = feature_importance_df[feature_importance_df['Importance'] > 0.01]
    feature_importance_df.sort_values(by='Importance', ascending=False, inplace=True)

    plt.figure(figsize=(10, 6))
    plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'])
    plt.xlabel('Importance')
    plt.ylabel('Feature')
    plt.title('Feature Importance in XGBoost Model')
    plt.gca().invert_yaxis()
    plt.show()

def plot_feature_importance(model, features, step, target_column):
    """
    Plots the feature importance for the given model.
    """
    feature_importance = model.feature_importances_
    feature_importance_df = pd.DataFrame({'Feature': features, 'Importance': feature_importance})
    feature_importance_df = feature_importance_df[feature_importance_df['Importance'] > 0.01]
    feature_importance_df.sort_values(by='Importance', ascending=False, inplace=True)

```

```
    plt.figure(figsize=(14, 7))
    bars = plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'], color='#78003F')

#       plt.barh(feature_importance_df['Feature'], feature_importance_df['Importance'])
    plt.xlabel('Importance')
    plt.ylabel('Feature')
    plt.title(f'Feature Importance in Step {step + 1} for {target_column}')

    # Adding percentage text at the end of each bar
    # Adding the percentage labels at the end of each bar
    for bar in bars:
        plt.text(bar.get_width(), bar.get_y() + bar.get_height() / 2, f'{bar.get_width():.2%}',
                 va='center', ha='left')
    plt.gca().invert_yaxis()
    plt.show()

def summarize_feature_importance(model, model_features, step):
    """
    Prints a summary of the feature importance for the given model.
    """
    feature_importance = model.feature_importances_
    feature_importance_df = pd.DataFrame({'Feature': model_features, 'Importance': feature_importance})
    feature_importance_df = feature_importance_df[feature_importance_df['Importance'] > 0.01]
    feature_importance_df.sort_values(by='Importance', ascending=False, inplace=True)
    feature_importance_df['Rank'] = range(1, len(feature_importance_df) + 1)
    feature_importance_df['Importance'] = feature_importance_df['Importance'].round(4)
    feature_importance_df.set_index('Rank', inplace=True)

    print(f"\nFeature Importance Summary at Step {step + 1}:")
    print(feature_importance_df)

def create_feature_importance_subplot(data_steps):
    """
    Creates a 4x2 subplot of horizontal bar charts to display feature importance for each step.

    Parameters:
    - data_steps (dict): A dictionary with steps as keys and lists of (feature, importance) tuples as values.
    """
    # Creating the subplot
    fig, axes = plt.subplots(4, 2, figsize=(15, 20))
    axes = axes.flatten()  # Flatten the axes array for easy iteration

    # Plotting data for each step
```

```
    for i, (step, values) in enumerate(data_steps.items()):
        ax = axes[i]
        df = pd.DataFrame(values, columns=["Feature", "Importance"])
        bars = ax.barh(df["Feature"], df["Importance"], color='#78003F')
        ax.set_title(f'{step}')
        ax.set_xlabel('Importance')
        ax.invert_yaxis()

        # Add text labels
        for index, bar in enumerate(bars):
            # Place the text inside the bar for the first bar, otherwise outside
            if index == 0:  # First bar
                text_x_pos = bar.get_width() / 1.15  # 2 Center of the bar
                text_color = 'white'
            else:
                text_x_pos = bar.get_width() + 0.005  # Slightly outside the bar
                text_color = 'black'

            ax.text(text_x_pos, bar.get_y() + bar.get_height() / 2,
                    f'{bar.get_width():.4f}', va='center', color=text_color, fontsize
=8)

    # Hide the empty subplot if the number of steps is odd
    if len(data_steps) % 2 != 0:
        axes[-1].axis('off')

    # Adjust layout
    plt.tight_layout()
    plt.show()

def find_best_rolling_windows_xgboost(data, target_column, date_column, lag_features,
    test_size, min_window=2, max_window=6):
    rolling_combinations = list(product(range(min_window, max_window + 1), repeat=len(
    lag_features)))
    results_list = []

    for combo in rolling_combinations:
        rolling_variables = {feature: window for feature, window in zip(lag_features.
    keys(), combo)}

        results = rolling_forecast_xgboost(
            data=data,
            target_column=target_column,
            date_column=date_column,
            lag_features=lag_features,
            test_size=test_size,
            rolling_variables=rolling_variables,
            plot = False
        )
```

```
        mae = mean_absolute_error(results['Actual'], results['Predictions'])
        print(f"Combination: {combo}, MAE: {mae}")
        results_list.append({
            'Combo': combo,
            'MAE': mae
        })

    results_df = pd.DataFrame(results_list)
    return results_df.sort_values(by='MAE')

#call XGBoost with  all features and their lags up to 12
lag_features = {
    'TextBlob_SI': list(range(0, 13)),
    'Vader_SI': list(range(0, 13)),
    'Transformers_SI': list(range(0, 13)),
    'Flair_SI': list(range(0, 13)),
    'Afinn_SI': list(range(0, 13)),
    'CCI': list(range(1, 13)),  # Assuming CCI is the target, start from lag 1
    'Average_wage': list(range(0, 13)),
    'Pension': list(range(0, 13)),
    'Inequality': list(range(0, 13)),
    'Inflation': list(range(0, 13)),
    'Unemployment_without_seasonality': list(range(0, 13)),
    'Unemployment_rate': list(range(0, 13))
}
rolling_variables = {}

forecast_results = rolling_forecast_xgboost(
    data=cci_data,
    target_column='CCI',
    date_column='Year_and_Month',
    lag_features=lag_features,
    test_size=7,
    rolling_variables=rolling_variables
)

#call XGBoost with  selected features and their lags
lag_features = {
    'CCI': [1],
    'Average_wage': [0],
    'Transformers_SI': [2],
    'Unemployment_without_seasonality':[0, 1],
}

rolling_variables = {}

forecast_results = rolling_forecast_xgboost(
    data=cci_data,
```

```
    target_column='CCI',
    date_column='Year_and_Month',
    lag_features=lag_features,
    test_size=7,
    rolling_variables=rolling_variables
)
#find the next rolling windows
# Example usage
lag_features = {
    'CCI': [1],
    'Average_wage': [0],
    'Transformers_SI': [2],
    'Unemployment_without_seasonality': [0, 1],
}

best_rolling_windows_df = find_best_rolling_windows_xgboost(
    data=cci_data,
    target_column='CCI',
    date_column='Year_and_Month',
    lag_features=lag_features,
    test_size=7,
    min_window=2,
    max_window=12
)

#call XGBoost with  selected features, their lags and rolling averages
lag_features = {
    'CCI': [1],
    'Average_wage': [0],
    'Transformers_SI': [2],
    'Unemployment_without_seasonality':[0, 1],
}

rolling_variables = {
    'CCI': 6,
    'Average_wage': 9,
    'Transformers_SI': 4,
    'Unemployment_without_seasonality': 4,
}

forecast_results = rolling_forecast_xgboost(
    data=cci_data,
    target_column='CCI',
    date_column='Year_and_Month',
    lag_features=lag_features,
    test_size=7,
    rolling_variables=rolling_variables
)
```