Vladimir Andreev

Macroeconomic Analysis of the US Economy

Do it yourself with AI

A Practical Guide to Data Science, Machine Learning and Deep Learning Projects

2025

This is a work of nonfiction intended to support students and professionals in the field of Artificial Intelligence. All efforts have been made to ensure the accuracy of the content at the time of publication.

For inquiries, contact: **eng.vladimir.andreev@gmail.com**

Table of Contents

Chapter 1: **Introduction**

This book is an upgrade of the author's course projects on artificial intelligence (AI): Data Science, Machine Learning and Deep Learning in the period 2024 – 2025.

It studies the macroeconomic indicators of the US economy, based mainly on data that is freely available from the Federal Reserve (FED). In the process of work, a study was conducted on the topic of AI and its application in the analysis of economic processes in the US economy. There are moments that are the author's creation and have no analogues on the Internet. At least, no such ones have been found that are freely available. Knowing what is on the Internet at the time of writing the book, there are demonstrably working models here that correctly show what they are intended for. With the necessary reliability of about 95%, which for AI and the relatively small data of about 10 – 12,000 records per collection (DataFrame) is an excellent achievement. Of course, over the years, data will accumulate, which will make the forecasts even more accurate, and the book more valuable.

Based on this, this book is a guide for AI students, researchers, investors, economists, financiers, traders, and in general for people who want to make their own "diagnosis" of the US economy. The latter still strongly influences all processes in the world, from economic and financial, through political, social to military.

Following the examples in the book, anyone who downloads current data for the US from the Internet can use AI tools to "dissect" their budget, make a recession forecast, and see what the state of the economy is, as well as its direction of development! And also see what the most critical moments in development are and when the "black swans" may arrive.

The book uses the latest data available at the time of writing for 2025, and it also includes a comparison with the results of AI models from previous periods. The Machine Learning (ML) and Deep Learning (DL) sections mainly deal with working with Time Series, due to the nature of the subject matter.

The easiest way to run the code from the book is to put it in a Jupyter Notebook. It is freely available with the Anaconda3 package. You can download it from here:

https://www.anaconda.com/download

You will need to install a few libraries, but their installation is not problematic and you can easily find out how to do this on the Internet. Enjoy your reading!

Important: I will leave this chapter with the "old data ", leaving you the pleasure of collecting the latest and doing your own analysis. The chapter with the Machine Learning project will be the same. There I will leave you to see my analysis - whether it corresponds to the events in the US economy in April 2025. And how accurate my forecast was. For the last chapter on Deep Learning, I will collect the latest data and you will have an up-to-date forecast.

Chapter 2: **Macroeconomic Analysis of the US Economy – part 1**

```
# import libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from matplotlib.ticker import FuncFormatter
from sklearn.feature_selection import mutual_info_regression
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

Date: August 2024, Data Science project

Abstract:

The purpose of this study is to determine the direction of change in the main macroeconomic indicators of the US economy (fundamental analysis). After finding the basic macroeconomic data, to trace the main trends. On the basis of this information, build a hypothesis and, after a detailed analysis of the movement of the most important macroeconomic data, draw the relevant conclusions.

1. Importance of Economic Indicators:

Economic indicators are the key statistics that show the direction of an economy. Important economic events determine price movements, so it is important to familiarize yourself with global economic events in order to perform proper **fundamental analysis** that will enable traders to make informed decisions.

Interpreting and analyzing the indicators is important for all investors as they indicate the overall health of the economy, anticipate its stability and enables investors to respond on time to sudden or unpredictable events, also known as economic shocks. They can also be referred to as a traders 'secret weapon' as they reveal what is to come next, what may be expected of the economy and which direction the markets can take. [1] Here are some of the most important economic indicators that this study will need:

1.1 GROSS DOMESTIC PRODUCT (GDP) - Level of Importance: High

The GDP report is one of the most important of all economic indicators, as it is the biggest measure of the overall state of the economy. It is the total of monetary value of all the goods and service produced by the entire economy during the quarter being measured (does not include international activity). The economic production and growth- what GDP represents, has a large impact on nearly everyone within that economy. For example, when the economy is healthy, what we will typically see is low unemployment and wage increases as businesses demand labor to meet the growing economy. A significant change in GDP, up or down, usually has a significant effect on the market, due to the fact that a bad economy usually means

lower earnings for companies, which translates into lower currency and stock prices. Investors really worry about negative GDP growth, which is one of the factors economists use to determine whether an economy is in recession.

1.2 CONSUMER PRICE INDEX (CPI) - Level of Importance: High

This report is the most widely used measure of inflation. It measures the change in the cost of a bundle of consumer goods and services from month to month. The base – year market basket of which he CPI is composed of, is derived from detailed expenditure information collected from thousands of families across the U.S. the basket consists of more than 200 categories of goods and services separated into eight groups: food and beverage, housing, apparel, transportation, medical care, recreation, education and communication and other goods and services. The extensive measures taken to formulate a clear picture of changes in the cost of living helps keep financial players to get a sense of inflation, which can destroy an economy if it is not controlled. Movements in the prices of goods and services most directly affect fixed-income securities (an investment that provides a return in the form of fixed periodic payments and the eventual return of principal in maturity). Modest and steady inflation is expected in a growing economy, but if the prices of resources used in production of good and services rise quickly, manufacturers may experience profit declines. On the other hand, deflation can be a negative sign indicating a decline in consumer demand.

The CPI is probably the most important and widely watched economic indicator and it is the best known measure for determining the cost of living changes. It is used to adjust wages, retirement benefits, tax brackets and other important economic indicators. It can tell the investors of what may happen in the financial markets, which share both direct and indirect relationships with consumer prices.

1.3 CONSUMER CONFIDENCE INDEX (CCI) - Level of Importance: High

As the name indicates, this indicator measure the consumer confidence. It is defined as the degree of optimism that the consumers have in terms of the state of the economy, which is expresses through consumers saving and spending activity. This economic indicator is released last Tuesday of the month, and it measures how confident people feel about their income stability that has a direct effect on their economic decisions, in other words, their spending activity. For this reason, CCI is seen as a key indicator for the overall shape of the economy.

The measurements are used as an indicative of consumption component level of the gross domestic product and the Federal Reserve looks at CCI when determining interest rate changes.

1.4 INTEREST RATES - Level of Importance: High

Interest rates are the major drivers of the forex market and all the above mentioned economic indicators are closely watched by the Federal Open Market Committee in order to determine the overall health of the economy. The FED can decide accordingly if they will lower, rise or leave the interest rates unchanged, all depending on the evidence gathered on the health of the economy. The existence of interest rates allows borrowers to spend money immediately instead of waiting to save the money to make a purchase. The lower the interest rate, the more willing people are to borrow money to make big purchases, such as houses or cars. When consumers pay less in interest, this gives them more money to spend which can create a ripple effect of increased spending throughout the economy. On the other hand, higher interest rates mean that consumers do not have as much disposable income and must cut back on spending. When higher interest rate are combined with increased lending standards, banks make fewer loans. This affects the consumers, businesses and farmers who will cut back on spending for new equipment, thus slowing the productivity or reducing the number of employees. Whenever interest rates are rising or falling, we hear about the federal funds rate (the rate banks use to lend each other money). The changes in the interest rates can affect both inflation and recession. Inflation refers to the rise in the price of goods and services over time, as a result of a strong and healthy economy. However, if inflation is left unchecked, it

can lead to a significant loss of purchasing power. As can be seen, interest rates affect the economy by influencing consumer and business spending, inflation and recessions.

1.5 TRADE BALANCE - Level of Importance: Medium

The Trade Balance is the difference between imports and exports of a given country for a given time period. It is used by economists as a statistical tool, as it enables them to understand the relative strength of a country's economy compared to other countries' economies and the flow of trade between nations.

Trade surpluses are desirable, where a positive value means that the exports are greater that imports; while on the other hand, trade deficits can lead towards a significant domestic debt. The index is published monthly.

2. Macroeconomic Data for the US Economy.

2.1 Criteria according to which the data for the present study were collected.

The time from **July 2022 to June 2024** was taken as the **base period of 2 years for data collection**. This is because the global economy as a whole during the period of COVID-19 operates on a non-market basis. From 2020 to the spring of 2022, many businesses and commercial establishments were closed - directly by regulations or indirectly (through restrictions imposed by health authorities). Billions of dollars and euros were poured into the economies of the world without cover. This resulted in a vicious combination of a closed economy and inflation.

If one traces the graph of the main world economic indicators, then for the period from 2020 to 2022 one will see an anomaly that would make any analysis very difficult and lead to wrong conclusions. Here is an example of such an anomaly [2]:

Find more statistics at Statista

For this reason, historical data of the main indicators is not taken, but **only the past period of 2 years is used**. Data on prices of precious and industrial metals will also be included to help the analysis. Much of the macroeconomic data comes out with a long time lag. Not all information will be possible to collect in the desired volume. However, the amount of data collected should be sufficient for analysis and subsequent conclusions about the development of the US economy in the future.

2.2 GROSS DOMESTIC PRODUCT (GDP)

From this site FEDERAL RESERVE BANK of ST. LOUIS we only download information for the period of our study: **Gross Domestic Product (GDP)**. [3] Units: **Billions of Dollars**, Seasonally Adjusted Annual Rate. Frequency: Quarterly.

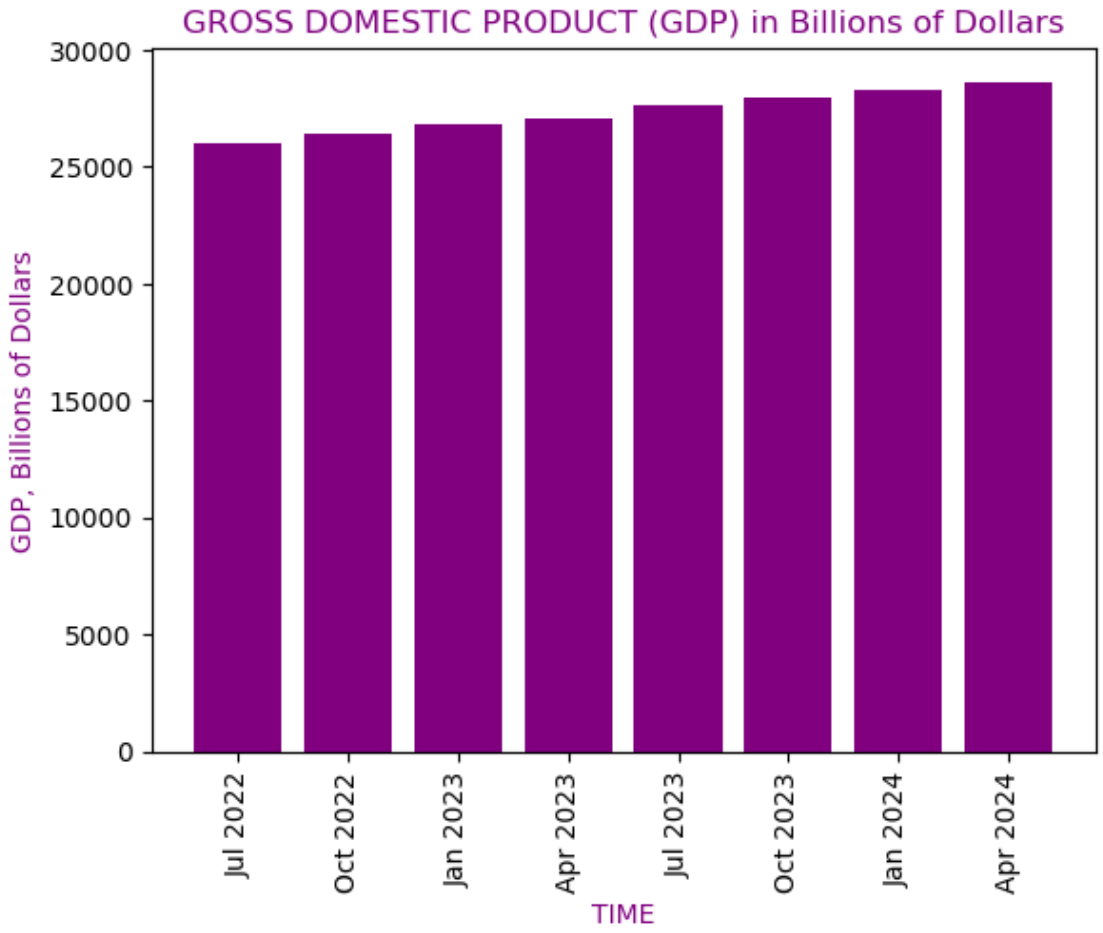
```
gdp = pd.read_csv('data/GDP.csv') # Read the CSV file.
gdp.columns = ["date", "gdp"]
gdp['gdp'] = gdp['gdp'] * 1_000_000_000 # Convert to dollars.
gdp
# Show all data: _______ DataFrame (DF) with real data! _______
```

	date	gdp
0	2022-07-01	2.599464e+13
1	2022-10-01	2.640840e+13
2	2023-01-01	2.681360e+13
3	2023-04-01	2.706301e+13
4	2023-07-01	2.761013e+13
5	2023-10-01	2.795700e+13
6	2024-01-01	2.826917e+13
7	2024-04-01	2.862915e+13

```
gdp.describe().T
```

```
     count          mean           std           min           25%  \
gdp    8.0  2.734314e+13  9.266329e+11  2.599464e+13  2.671230e+13

               50%           75%           max
gdp  2.733657e+13  2.803504e+13  2.862915e+13
```

```
# A copy of the table in order to correctly represent the data.
gdp_for_table = gdp.copy()

# Split the date into day, month, and year.
gdp_for_table['year'] = pd.to_datetime(gdp_for_table['date']).dt.year
gdp_for_table['month'] = pd.to_datetime(gdp_for_table['date']).dt.month

# Create custom labels with just month and year.
date_labels = gdp_for_table['date'].apply(lambda x: f"{pd.to_datetime(x).strfti
me('%b %Y')}")

# Convert to Billions of Dollars.
gdp_for_table['gdp'] = gdp_for_table['gdp'] / 1_000_000_000

plt.bar(gdp_for_table.index, gdp_for_table['gdp'], color = "purple")
# Display month and year as labels.
plt.xticks(gdp_for_table.index, date_labels, rotation = 90)
plt.title("GROSS DOMESTIC PRODUCT (GDP) in Billions of Dollars", color = "purpl
e")
plt.xlabel("TIME", color = "purple")
plt.ylabel("GDP, Billions of Dollars", color = "purple")
plt.show()
```



Data for the period under study are given by **quartals**, not by months. To obtain the requested data from **July 2022** to **June 2024**, we use a **linear approximation**.

```
gdp['date'] = pd.to_datetime(gdp['date'])
gdp_approx = gdp.copy()

# Create a new DataFrame with all dates from 2022-07-01 to 2024-06-01.
all_dates = pd.date_range(start='2022-07-01', end='2024-06-01', freq='MS')
gdp_all_dates = pd.DataFrame({'date': all_dates})

# Merge the two DataFrames and interpolate.
gdp_approx = pd.merge(gdp_all_dates, gdp, on='date', how='left')

# Linear interpolation.
gdp_approx['gdp'] = gdp_approx['gdp'].interpolate(method='linear')

# Remove the 'year' and 'month' columns.
if 'year' in gdp_approx.columns:
    gdp_approx = gdp_approx.drop(columns=['year'])
if 'month' in gdp_approx.columns:
    gdp_approx = gdp_approx.drop(columns=['month'])

gdp_approx.head() # Show first rows only...
```

```
        date           gdp
0 2022-07-01  2.599464e+13
1 2022-08-01  2.613256e+13
2 2022-09-01  2.627048e+13
```

```
3 2022-10-01  2.640840e+13
4 2022-11-01  2.654347e+13
```

2.3 CONSUMER PRICE INDEX (CPI)

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we only download information for the period of our study: **Consumer Price Index (CPI)** for All Urban Consumers, All Items in U.S. City Average (CPIAUCSL). [4]

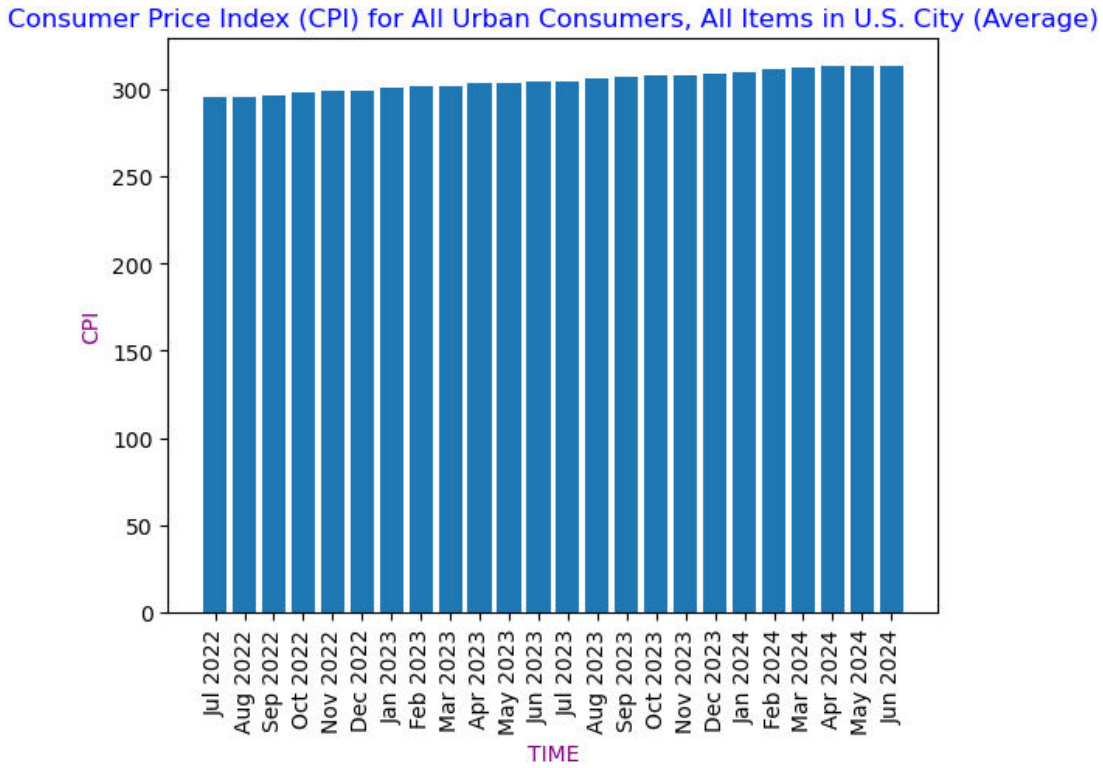
```
cpi = pd.read_csv('data/Consumer Price Index.csv') # Read the CSV file.
cpi.columns = ["date", "cpi"]
cpi.head() # Show first rows only...
```

```
         date      cpi
0  2022-07-01  294.977
1  2022-08-01  295.209
2  2022-09-01  296.341
3  2022-10-01  297.863
4  2022-11-01  298.648
```

```
# Returns statistical description of the data in the DataFrame.

cpi[['cpi']].describe().T
```

```
     count        mean       std      min     25%       50%        75%  \
cpi   24.0  304.612875  5.883572  294.977  299.97  304.3155  308.97775

         max
cpi  313.225
```

```
# Split the date into day, month, and year.
cpi['year'] = pd.to_datetime(cpi['date']).dt.year
cpi['month'] = pd.to_datetime(cpi['date']).dt.month

# Create custom labels with just month and year.
date_labels = cpi['date'].apply(lambda x: f"{pd.to_datetime(x).strftime('%b %Y'
)}")
# Display month and year as vertical labels.
plt.bar(cpi.index, cpi['cpi'])
plt.xticks(cpi.index, date_labels, rotation = 90)
plt.title("Consumer Price Index (CPI) for All Urban Consumers, All Items in U.S
. City (Average)", color = "blue")
plt.xlabel("TIME", color = "purple")
plt.ylabel("CPI", color = "purple")
plt.show()
```



2.4 CONSUMER CONFIDENCE INDEX (CCI)

From this site MacroVar we download information: **CONSUMER CONFIDENCE INDEX (CCI)**, United States Conference Board Consumer Confidence Analytics & Data. [5] Since the data is from 2014, we filter only those that are for our studied period.

```
cci = pd.read_csv('data/CCI.csv') # Read the CSV file.
cci.columns = ["date", "cci"]
# Convert the "date" column to datetime format.
cci['date'] = pd.to_datetime(cci['date'])
# Filter the data for the period from 2022 to 2024.
filtered_cci = cci[(cci['date'] >= '2022-07-01') & (cci['date'] <= '2024-06-30'
)]
del cci
filtered_cci = filtered_cci.reset_index(drop=True)
filtered_cci.head() # Show first rows only...
```

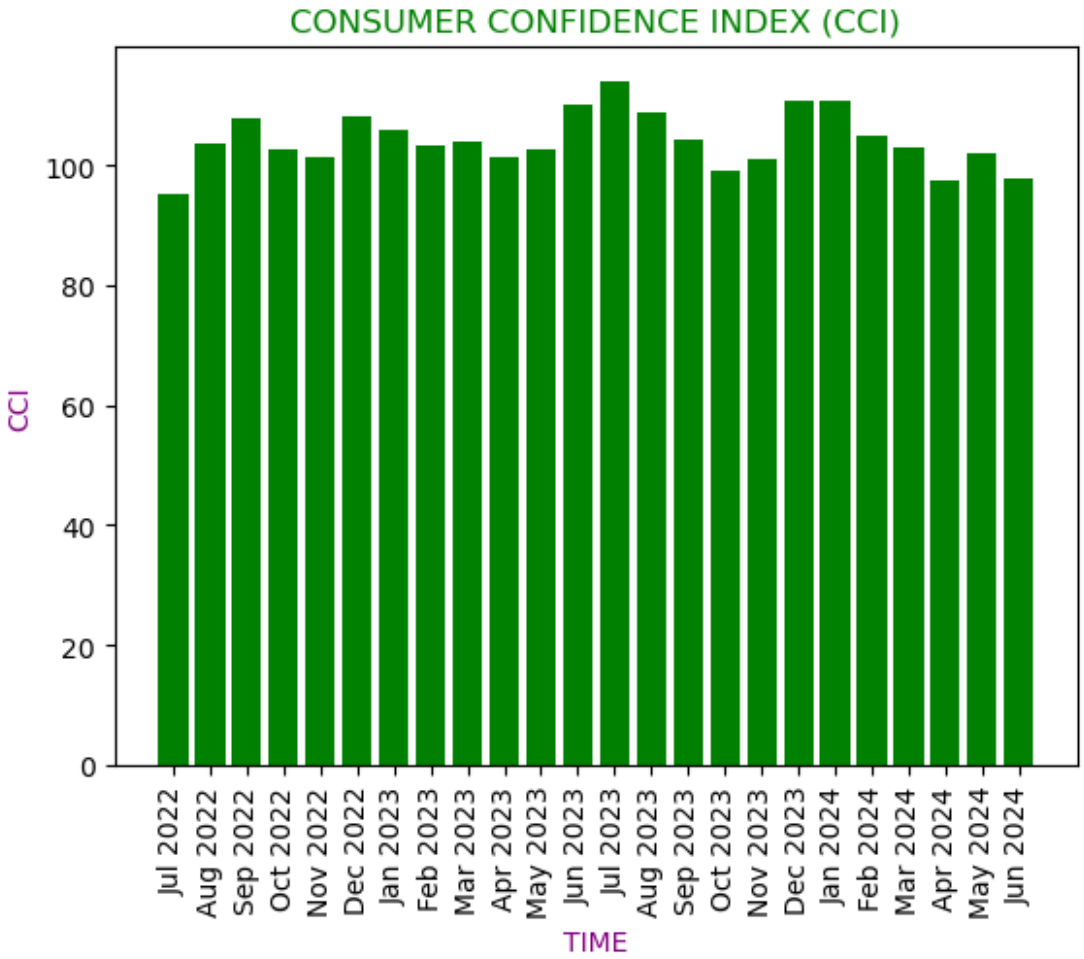
```
        date    cci
0 2022-07-31   95.3
1 2022-08-31  103.6
2 2022-09-30  107.8
3 2022-10-31  102.5
4 2022-11-30  101.4
```

```
filtered_cci[['cci']].describe().T # .T only for DataFrame, not for Series.
```

```
     count        mean       std   min      25%    50%      75%    max
cci   24.0  104.170833  4.671977  95.3  101.375  103.5  107.925  114.0
```

```
# Split the date into day, month, creating new columns in the DataFrame.
filtered_cci = filtered_cci.assign(year=pd.to_datetime(filtered_cci['date']).dt
.year)
filtered_cci = filtered_cci.assign(month=pd.to_datetime(filtered_cci['date']).d
```

```
t.month)

# Create custom labels with just month and year.
date_labels = filtered_cci['date'].apply(lambda x: f"{pd.to_datetime(x).strftim
e('%b %Y')}")

plt.bar(filtered_cci.index, filtered_cci['cci'], color = "green")

# Display month and year as vertical labels.
plt.xticks(filtered_cci.index, date_labels, rotation=90)
plt.title("CONSUMER CONFIDENCE INDEX (CCI)", color = "green")
plt.xlabel("TIME", color="purple")
plt.ylabel("CCI", color="purple")
plt.show()
```



2.5 INTEREST RATES

From this site FEDERAL RESERVE BANK of ST. LOUIS we only download information for the period of our study: **Federal Funds Effective Rate (FEDFUNDS)**. [6] The federal funds rate is the interest rate at which depository institutions trade federal funds (balances held at Federal Reserve Banks) with each other overnight. When a depository institution has surplus balances in its reserve account, it lends to other banks in need of larger balances. In simpler terms, a bank with excess cash, which is often referred to as liquidity, will lend to another bank that needs to quickly raise liquidity.

1. The rate that the borrowing institution pays to the lending institution is determined between the two banks; the weighted average rate for all of these types of negotiations is called the effective federal funds rate.

2. The effective federal funds rate is essentially determined by the market but is influenced by the Federal Reserve through open market operations to reach the federal funds rate target.

The Federal Open Market Committee (FOMC) meets eight times a year to determine the federal funds target rate. As previously stated, this rate influences the effective federal funds rate through open market operations or by buying and selling of government bonds (government debt).

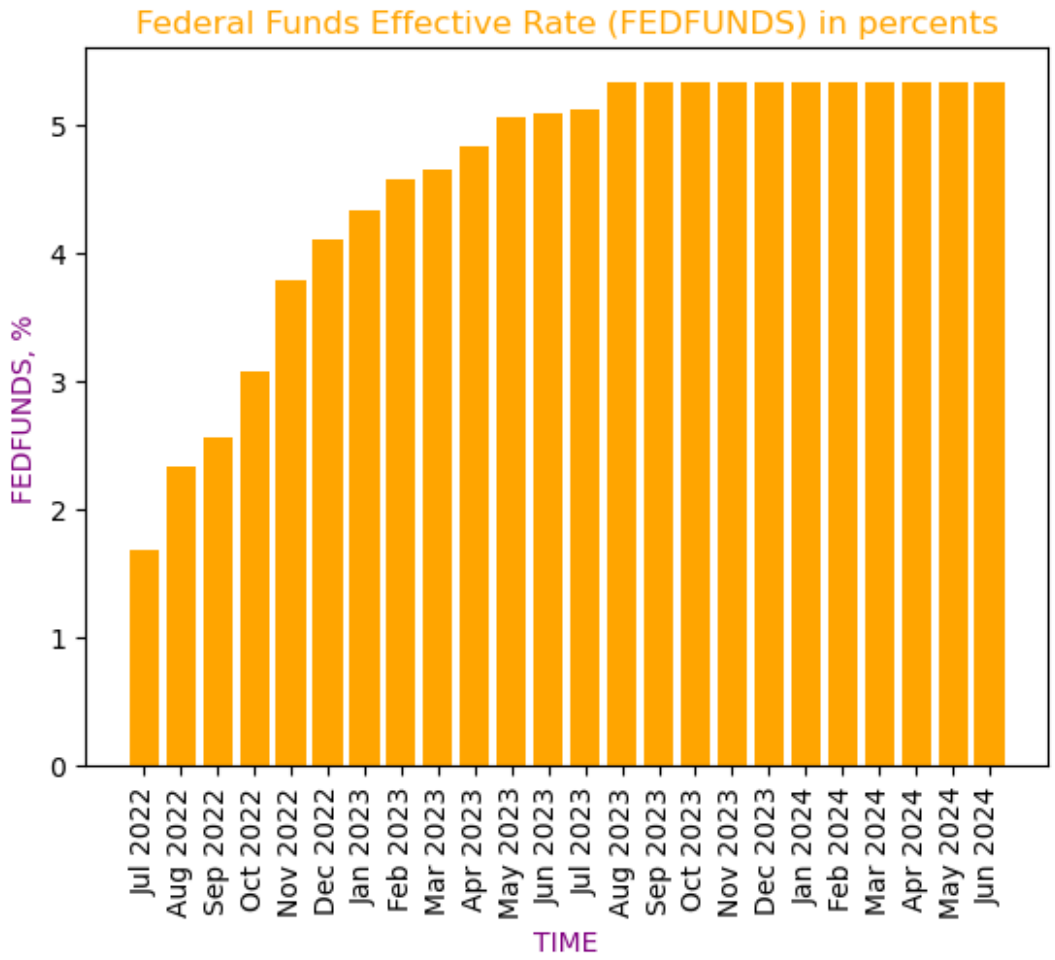
```
fed_funds = pd.read_csv('data/FEDFUNDS.csv') # Read the CSV file.
fed_funds.columns = ["date", "fed_funds"]
fed_funds.head()
# ____ Display data in percentages!! _________ Show first rows only...
```

```
         date  fed_funds
0  2022-07-01       1.68
1  2022-08-01       2.33
2  2022-09-01       2.56
3  2022-10-01       3.08
4  2022-11-01       3.78
```

```
fed_funds.describe().T
```

```
           count   mean       std   min     25%  50%   75%   max
fed_funds   24.0  4.575  1.098659  1.68  4.2725  5.1  5.33  5.33
```

```
# Split the date into day, month, and year.
fed_funds['year'] = pd.to_datetime(cpi['date']).dt.year
fed_funds['month'] = pd.to_datetime(cpi['date']).dt.month

# Create custom labels with just month and year.
date_labels = fed_funds['date'].apply(lambda x: f"{pd.to_datetime(x).strftime('
%b %Y')}")

plt.bar(fed_funds.index, fed_funds['fed_funds'], color = "orange")

# Display month and year as vertical labels.
plt.xticks(fed_funds.index, date_labels, rotation = 90)
plt.title("Federal Funds Effective Rate (FEDFUNDS) in percents", color = "orang
e")
plt.xlabel("TIME", color = "purple")
plt.ylabel("FEDFUNDS, %", color = "purple")
plt.show()
```



2.6 TRADE BALANCE

From this site FEDERAL RESERVE BANK of ST. LOUIS we only download information for the period of our study: **Trade Balance: Goods, Balance of Payments Basis (BOPGTB)**, Units: **Millions of Dollars**, Seasonally Adjusted. [7]

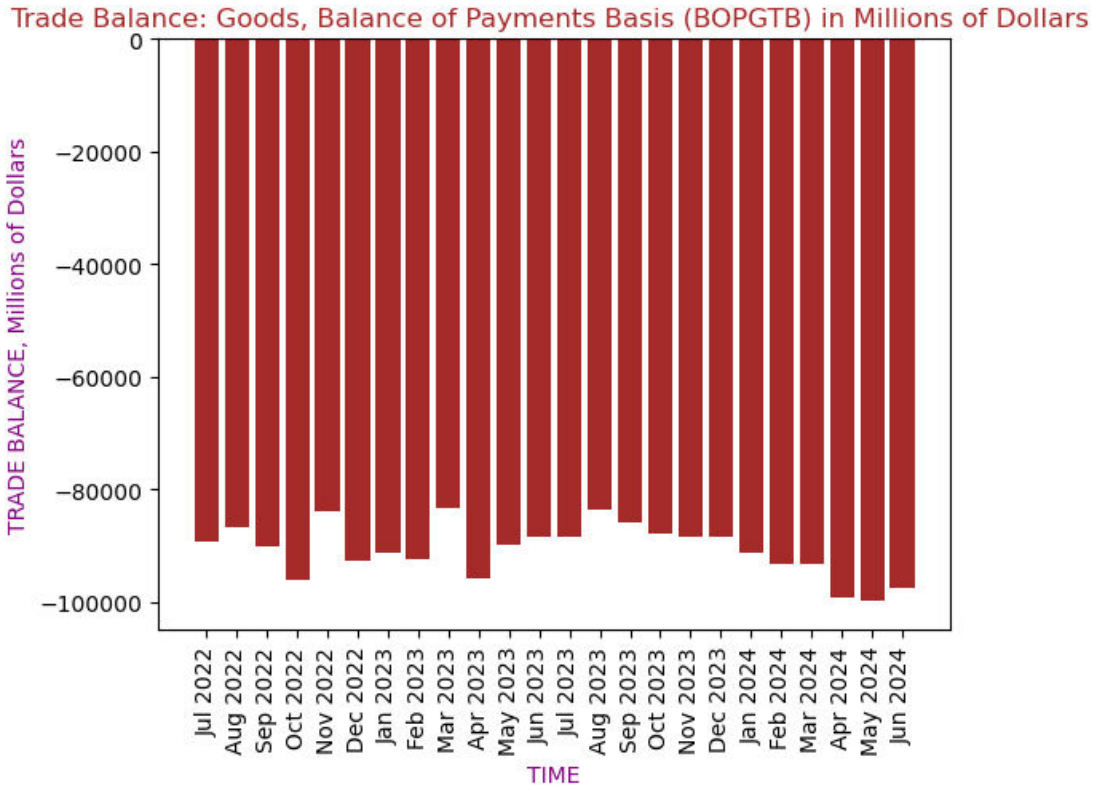
```
tb = pd.read_csv('data/BOPGTB.csv') # Read the CSV file.
tb.columns = ["date", "trade_balance"]
# Convert to dollars!!
tb['trade_balance'] = tb['trade_balance'] * 1_000_000
tb.head()
# __________ DF with real data! __________ Show first rows only...
```

```
         date  trade_balance
0  2022-07-01   -8.923600e+10
1  2022-08-01   -8.678200e+10
2  2022-09-01   -9.001500e+10
3  2022-10-01   -9.614500e+10
4  2022-11-01   -8.397500e+10
```

```
tb.describe().T
```

```
                count          mean           std           min           25%  \
trade_balance    24.0 -9.068129e+10  4.670357e+09 -9.984300e+10 -9.322400e+10

                      50%           75%           max
trade_balance -8.990900e+10 -8.821825e+10 -8.314900e+10
```

```
# A copy of the table in order to correctly represent the data.

tb_for_table = tb.copy()
# To convert to millions of dollars!!
tb_for_table['trade_balance'] = tb_for_table['trade_balance'] / 1_000_000

# Split the date into day, month, and year.
tb_for_table['year'] = pd.to_datetime(tb_for_table['date']).dt.year
tb_for_table['month'] = pd.to_datetime(tb_for_table['date']).dt.month

# Create custom labels with just month and year.
date_labels = tb['date'].apply(lambda x: f"{pd.to_datetime(x).strftime('%b %Y')
}")

plt.bar(tb_for_table.index, tb_for_table['trade_balance'], color = "brown")

# Display month and year as vertical labels.
plt.xticks(tb_for_table.index, date_labels, rotation = 90)
plt.title("Trade Balance: Goods, Balance of Payments Basis (BOPGTB) in Millions
of Dollars ", color = "brown")
plt.xlabel("TIME", color = "purple")
plt.ylabel("TRADE BALANCE, Millions of Dollars", color = "purple")
plt.show()
```



2.7 US DEBT

From this site FiscalData we download information: **U.S. Treasury Monthly Statement of the Public Debt (MSPD)**, Date Range (Record Date): 5 Years. **Debt in Millions of Dollars!** [8]

The U.S. Treasury Monthly Statement of the Public Debt (MSPD) dataset details the Treasury's outstanding debts and the statutory debt limit. Debt is categorized by whether it is marketable or non-marketable and whether it is debt held by the public or debt held by government agencies. All amounts are reported in

millions of U.S. dollars. Data is published on the fourth business day of each month, detailing the debt as of the end of the previous month.

```
# Read the dataset using the compression zip.
debt = pd.read_csv('data/MSPD_SumSecty.zip',compression='zip')

# We leave only the requested columns...
keep_col = ['Record Date', 'Security Type Description', 'Total Public Debt Outs
tanding (in Millions)']
debt_filtered = debt[keep_col]

# We leave only the requested dates...
debt_filtered = debt_filtered[(debt_filtered['Record Date'] >= '2022-07-01') &
(debt_filtered['Record Date'] <= '2024-06-31')]

# selecting rows based on condition.
debt_filtered = debt_filtered.loc[debt_filtered['Security Type Description'] ==
'Total Public Debt Outstanding']

# Rename columns.
debt_filtered.columns = ["date", "type", 'total_debt']

debt_filtered.head() # Show first rows only...
```

	date	type	total_debt
13	2024-06-30	Total Public Debt Outstanding	3.483163e+07
27	2024-05-31	Total Public Debt Outstanding	3.466712e+07
41	2024-04-30	Total Public Debt Outstanding	3.461699e+07
55	2024-03-31	Total Public Debt Outstanding	3.458653e+07
69	2024-02-29	Total Public Debt Outstanding	3.447108e+07

```
del debt # Drop debt table to free memory.
debt_filtered['total_debt'].dtype # float64 to hold such large numbers.
# No need to change the data type.
```

```
dtype('float64')
```

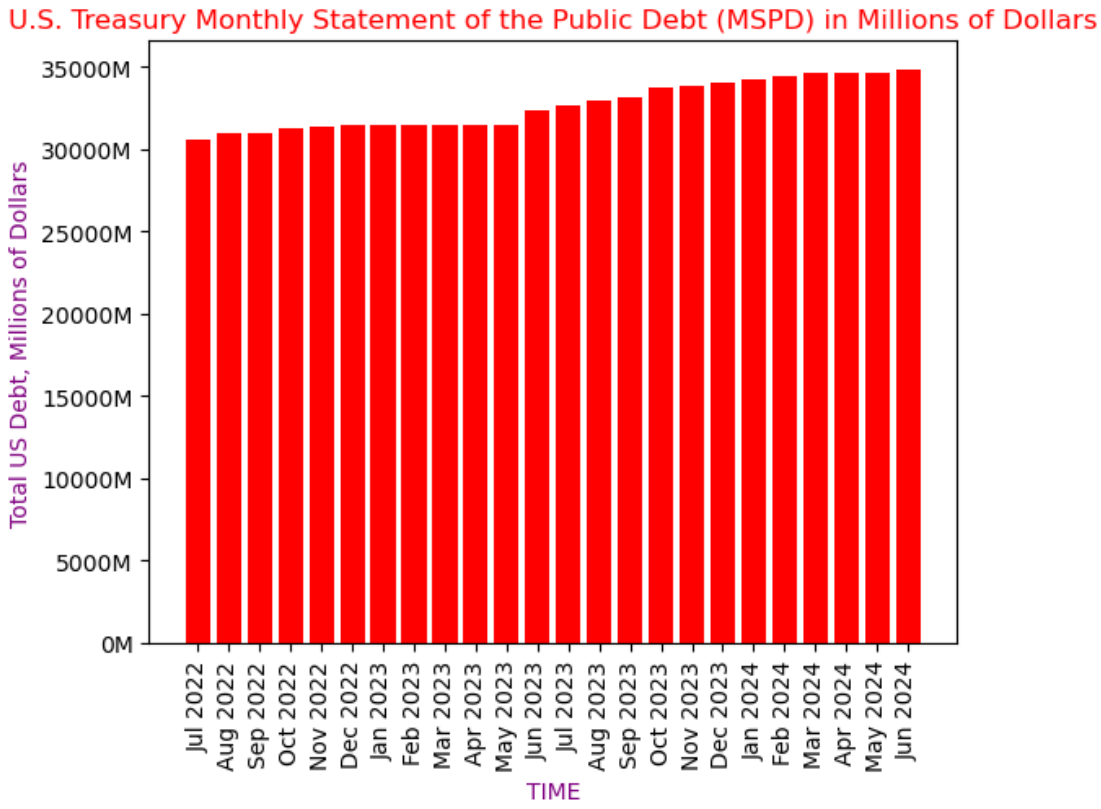
```
# Create DataFrame.
debt_real = pd.DataFrame(debt_filtered)

# Convert date column to datetime type.
debt_real['date'] = pd.to_datetime(debt_real['date'])

# Sort a DataFrame by date in ascending order.
debt_real = debt_real.sort_values(by='date', ascending=True)

# Reindex the DataFrame.
debt_real.reset_index(drop=True, inplace=True)

# Convert millions of dollars to dollars!
debt_real['total_debt'] = debt_real['total_debt'] * 1_000_000

debt_real.head()
# Show first rows only... __________ DF with real data! __________
```

```
        date                              type    total_debt
0 2022-07-31  Total Public Debt Outstanding  3.059511e+13
1 2022-08-31  Total Public Debt Outstanding  3.093608e+13
2 2022-09-30  Total Public Debt Outstanding  3.092891e+13
3 2022-10-31  Total Public Debt Outstanding  3.123830e+13
4 2022-11-30  Total Public Debt Outstanding  3.141332e+13
```

```
debt_real[['total_debt']].describe().T
```

```
            count          mean           std           min           25%  \
total_debt   24.0  3.265804e+13  1.463557e+12  3.059511e+13  3.144616e+13

                     50%           75%           max
total_debt  3.247043e+13  3.404891e+13  3.483163e+13
```

```
# A copy of the table in order to correctly represent the data.

debt_for_table = debt_real.copy()
# Formatting the Y-axis for a new dollar scale.
def millions(x, pos):
    'The two args are the value and tick position'
    return '%.0fM' % (x * 1e-9)
formatter = FuncFormatter(millions)
fig, ax = plt.subplots()
ax.yaxis.set_major_formatter(formatter)

# Split the date into day, month, and year.
debt_for_table['year'] = pd.to_datetime(debt_for_table['date']).dt.year
debt_for_table['month'] = pd.to_datetime(debt_for_table['date']).dt.month

# Create custom labels with just month and year.
date_labels = debt_for_table['date'].apply(lambda x: f"{pd.to_datetime(x).strft
ime('%b %Y')}")

plt.bar(debt_for_table.index, debt_for_table['total_debt'], color = "red")

# Time as vertical label.
plt.xticks(debt_for_table.index, date_labels, rotation = 90)

plt.title("U.S. Treasury Monthly Statement of the Public Debt (MSPD) in Million
s of Dollars", color = "red")
plt.xlabel("TIME", color = "purple")
plt.ylabel("Total US Debt, Millions of Dollars", color = "purple")
plt.show()

# 3.466712e+07 in Millions of Dollars => 3.466712e+13 => 34.66712e+12 =>
# 34.66712 trillions USD => 34 667.12 millions USD
```



2.8 UNEMPLOYMENT

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we only download information for the period of our study: Infra-Annual Labor Statistics: **Monthly Unemployment Rate**. [9] Units: **Percent**, Seasonally Adjusted. Frequency: Monthly.

```
unemployment = pd.read_csv('data/LRHUTTTTUSM.csv')
unemployment.columns = ["date", "unemployment"]
unemployment.head()
# ____ Display data in percentages!! ______ Show first rows only...
```

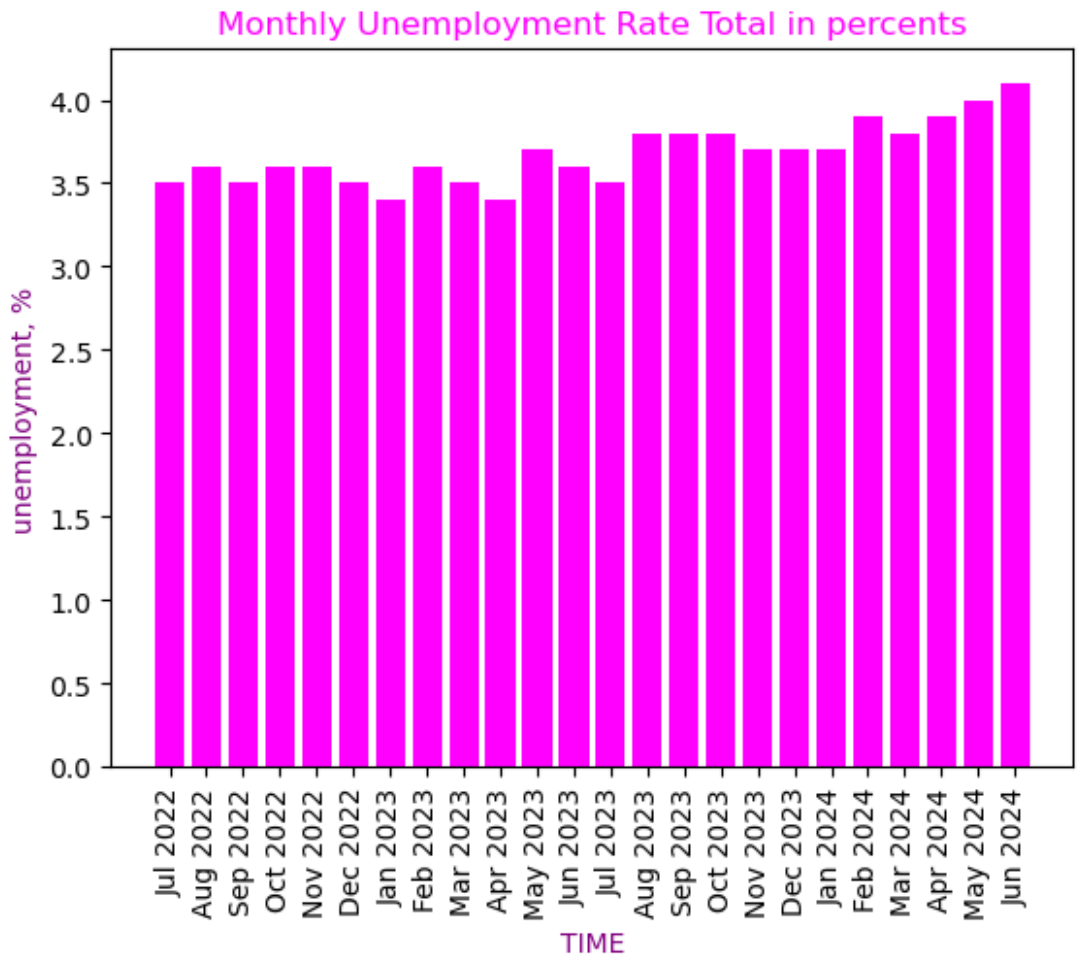
```
         date  unemployment
0  2022-07-01           3.5
1  2022-08-01           3.6
2  2022-09-01           3.5
3  2022-10-01           3.6
4  2022-11-01           3.6
```

```
unemployment.describe().T
```

```
              count   mean       std  min  25%   50%  75%  max
unemployment   24.0  3.675  0.184744  3.4  3.5  3.65  3.8  4.1
```

```
# Split the date into day, month, and year.
unemployment['year'] = pd.to_datetime(unemployment['date']).dt.year
unemployment['month'] = pd.to_datetime(unemployment['date']).dt.month

# Create custom labels with just month and year.
date_labels = unemployment['date'].apply(lambda x: f"{pd.to_datetime(x).strftim
e('%b %Y')}")

plt.bar(unemployment.index, unemployment['unemployment'], color = "magenta")
```

```
plt.xticks(unemployment.index, date_labels, rotation = 90)
plt.title("Monthly Unemployment Rate Total in percents", color = "magenta")
plt.xlabel("TIME", color = "purple")
plt.ylabel("unemployment, %", color = "purple")
plt.show()
```



2.9 OTHER ECONOMIC INDICATORS

From this site: FiscalData, we pull the **US budget data** for the last few years **by spending categories** [10]. Already on the site, we filter the requested information, and then **only 2023 and 2023** will be taken. This is because the budget year in the USA starts in September, and the data on its implementation comes out with a big delay. Monitoring the implementation of the budget by month would not give much result.

```
# Read the dataset using the compression zip.
budget = pd.read_csv('data/USFR_StmtNetCost.zip', compression='zip')

budget.head() # Show first rows only...
```

```
  Record Date  Statement Fiscal Year  \
0  2023-09-30                   2023
1  2023-09-30                   2023
2  2023-09-30                   2023
3  2023-09-30                   2023
4  2023-09-30                   2023
```

```
                                        Agency Name  Net Cost (in Billions)  \
0            Department of Health and Human Services                  1714.1
1                      Department of Veterans Affairs                  1455.3
2                      Social Security Administration                  1432.8
3                               Department of Defense                  1003.3
4  Interest on Treasury Securities held by the pu...                   678.0

   Source Line Number  Fiscal Year
0                   1         2023
1                   2         2023
2                   3         2023
3                   4         2023
4                   5         2023
```

```
budget.shape # 422 rows, 6 columns
```

```
(422, 6)
```

```
# Keep only the required columns.
keep_col = ['Statement Fiscal Year', 'Agency Name', 'Net Cost (in Billions)', '
Fiscal Year']
budget_filtered = budget[keep_col]

# Renaming columns.
budget_filtered.columns = ['statement_date', 'agency', 'net_cost', 'date']

# Convert values to dollars.
budget_filtered.loc[:, 'net_cost'] = budget_filtered['net_cost']*1_000_000_000
```

```
budget_filtered.dtypes
```

```
statement_date      int64
agency             object
net_cost          float64
date                int64
dtype: object
```

```
# Filter by year (2022 or 2023).
filtered_by_year = budget_filtered[(budget_filtered['date'] == 2022) | (budget_
filtered['date'] == 2023)]

# Filter by matching year in 'date' and 'statement_date'.
filtered_by_date_match = filtered_by_year[filtered_by_year['statement_date'] ==
filtered_by_year['date']]

# 'budget_filtered' will now contain only rows that meet both conditions.
budget_filtered = filtered_by_date_match

# Remove empty lines and duplicate lines.
cleaned_data = budget_filtered.dropna()
cleaned_data = cleaned_data.drop_duplicates()

# Remove rows with "Total" and "Subtotal" in 'agency'.
```

```
cleaned_data = cleaned_data[~cleaned_data['agency'].isin(['Total', 'Subtotal'])
]

# Remove columns with missing data.
cleaned_data = cleaned_data.dropna(axis=1, how='any')
budget_filtered = cleaned_data # Update!

budget_filtered
```

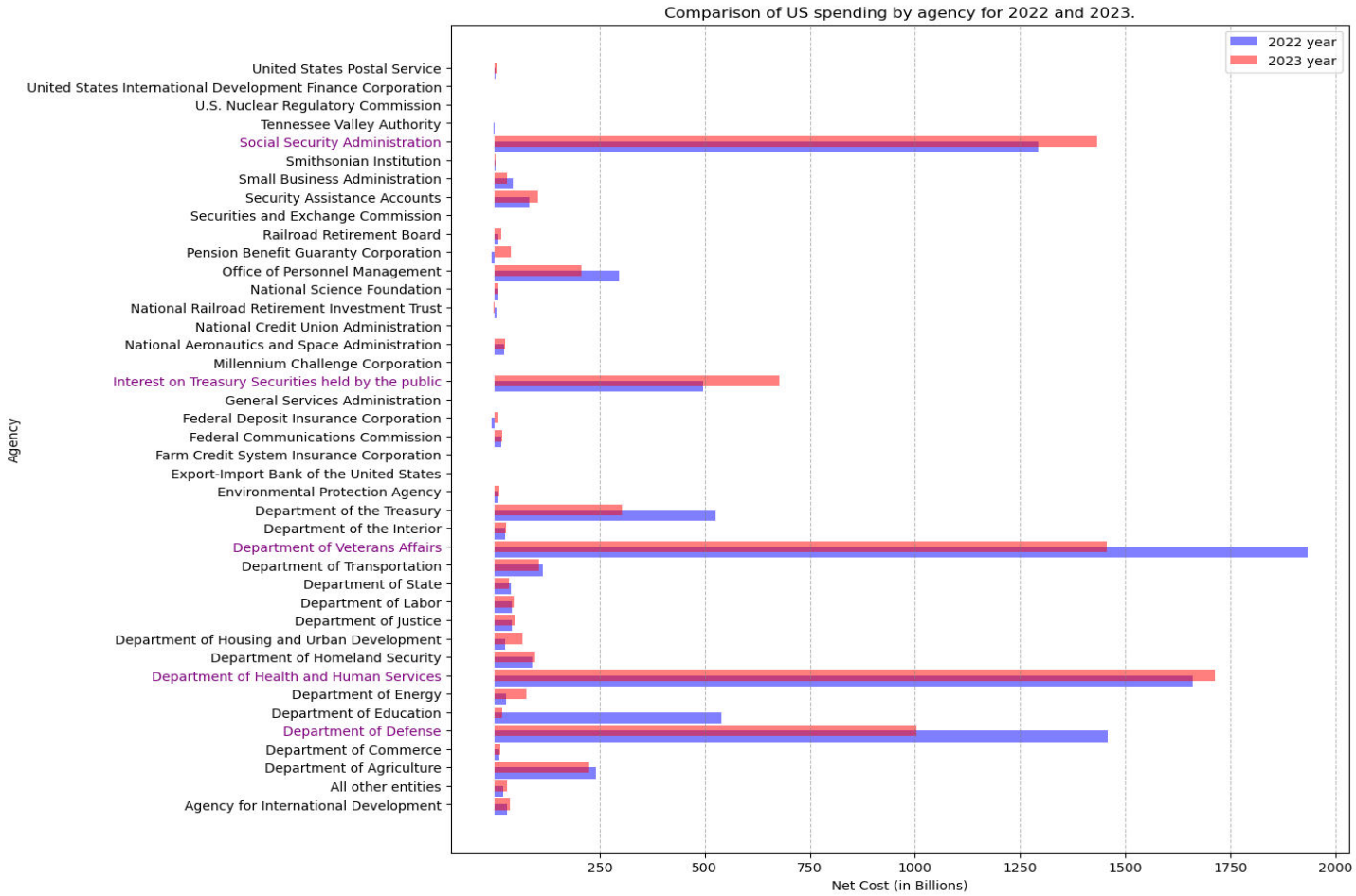
```
     statement_date                                             agency  \
0              2023           Department of Health and Human Services
1              2023                     Department of Veterans Affairs
2              2023                     Social Security Administration
3              2023                               Department of Defense
4              2023  Interest on Treasury Securities held by the pu...
..              ...                                                ...
122            2022          Farm Credit System Insurance Corporation
123            2022                           Tennessee Valley Authority
124            2022             Federal Deposit Insurance Corporation
125            2022              Pension Benefit Guaranty Corporation
126            2022                                 All other entities

         net_cost  date
0    1.714100e+12  2023
1    1.455300e+12  2023
2    1.432800e+12  2023
3    1.003300e+12  2023
4    6.780000e+11  2023
..            ...   ...
122 -6.000000e+08  2022
123 -1.100000e+09  2022
124 -6.300000e+09  2022
125 -7.000000e+09  2022
126  2.020000e+10  2022

[82 rows x 4 columns]
```

```
# Calculation of the total net_cost for each category in the agency.
sum_net_cost = budget_filtered.groupby('agency')['net_cost'].sum()

# Selecting the first 5 categories with the largest net_cost.
top_5_agencies = sum_net_cost.nlargest(5).index

# Filter the data for these 5 categories.
top_5_data = budget_filtered[budget_filtered['agency'].isin(top_5_agencies)]

# Grouping and calculating the descriptive statistics for the net_cost column
for the top 5 categories.
desc_stats = top_5_data.groupby('agency')['net_cost'].describe()

# Show only the top 5 agencies with the highest net_cost.
desc_stats_top_5 = desc_stats.nlargest(5, 'mean')
desc_stats_top_5
```

```
                                                 count          mean  \
agency
Department of Veterans Affairs                     2.0  1.695100e+12
Department of Health and Human Services            2.0  1.687050e+12
Social Security Administration                     2.0  1.363450e+12
Department of Defense                              2.0  1.231150e+12
Interest on Treasury Securities held by the public 2.0  5.872500e+11

                                                        std  \
agency
Department of Veterans Affairs                  3.391284e+11
Department of Health and Human Services         3.825448e+10
Social Security Administration                  9.807571e+10
Department of Defense                           3.222286e+11
Interest on Treasury Securities held by the public  1.283399e+11

                                                        min  \
agency
Department of Veterans Affairs                  1.455300e+12
Department of Health and Human Services         1.660000e+12
Social Security Administration                  1.294100e+12
Department of Defense                           1.003300e+12
Interest on Treasury Securities held by the public  4.965000e+11

                                                        25%  \
agency
Department of Veterans Affairs                  1.575200e+12
Department of Health and Human Services         1.673525e+12
Social Security Administration                  1.328775e+12
Department of Defense                           1.117225e+12
Interest on Treasury Securities held by the public  5.418750e+11

                                                        50%  \
agency
Department of Veterans Affairs                  1.695100e+12
Department of Health and Human Services         1.687050e+12
Social Security Administration                  1.363450e+12
Department of Defense                           1.231150e+12
Interest on Treasury Securities held by the public  5.872500e+11

                                                        75%           max
agency
Department of Veterans Affairs                  1.815000e+12  1.934900e+12
Department of Health and Human Services         1.700575e+12  1.714100e+12
Social Security Administration                  1.398125e+12  1.432800e+12
Department of Defense                           1.345075e+12  1.459000e+12
Interest on Treasury Securities held by the public  6.326250e+11  6.780000e+11
```

```
# Filter data for 2022 and calculate sum net_cost for each agency.
sum_net_cost_2022 = budget_filtered[budget_filtered['date'] == 2022].groupby('a
gency')['net_cost'].sum()

# Filter data for 2023 and calculate sum net_cost for each agency.
sum_net_cost_2023 = budget_filtered[budget_filtered['date'] == 2023].groupby('a
```

```
gency')['net_cost'].sum()

# Create a horizontal bar plot.
plt.figure(figsize=(12, 12))

bar_width = 0.6  # Adjust the width of the bars.
bar_positions_2022 = np.arange(len(sum_net_cost_2022))
bar_positions_2023 = np.arange(len(sum_net_cost_2023)) + (bar_width / 2)

plt.barh(bar_positions_2022, sum_net_cost_2022.values / 1_000_000_000, bar_widt
h, color='b', alpha=0.5, label='2022 year')
plt.barh(bar_positions_2023, sum_net_cost_2023.values / 1_000_000_000, bar_widt
h, color='r', alpha=0.5, label='2023 year')

plt.xlabel('Net Cost (in Billions)')
plt.ylabel('Agency')
plt.title('Comparison of US spending by agency for 2022 and 2023.')

plt.yticks(bar_positions_2022 + bar_width / 2, sum_net_cost_2022.index)

# Set the X axis readings.
plt.xticks([250, 500, 750, 1000, 1250, 1500, 1750, 2000])

# Add the vertical gray lines.
plt.grid(axis='x', alpha=0.5, color='gray', linestyle='--')
# Coloring the text of the top 5 categories.
for label in plt.gca().get_yticklabels():
    if label.get_text() == 'Department of Defense':
        label.set_color('purple')
    elif label.get_text() == 'Department of Veterans Affairs':
        label.set_color('purple')
    elif label.get_text() == 'Social Security Administration':
        label.set_color('purple')
    elif label.get_text() == 'Department of Health and Human Services':
        label.set_color('purple')
    elif label.get_text() == 'Interest on Treasury Securities held by the publi
c':
        label.set_color('purple')

plt.legend()
plt.show()
```




```
# Total for the entire year 2023.
total_budget_2023 = sum_net_cost_2023.sum()

# Calculation of the percentage share of the departments.
agencies_of_interest = [
    'Department of Defense',
    'Department of Veterans Affairs',
    'Interest on Treasury Securities held by the public',
    'National Science Foundation',
    'Department of Transportation',
    'Department of Education',
    'Department of Energy'
]
# Calculation:
percentages = {}
for agency in agencies_of_interest:
    if agency in sum_net_cost_2023:
        percentages[agency] = (sum_net_cost_2023[agency] / total_budget_2023) *
100
    else:
        percentages[agency] = 0

# Print results with spaces.
def format_number(number):
    return f"{number:,.2f}".replace(",", " ")

def bold_text(text):
```

```
    return f"\033[1m{text}\033[0m"

print(bold_text(f"Total spending for 2023: ${format_number(total_budget_2023)}"
))
print("Percentages of the total spending for 2023:")
for agency, percentage in percentages.items():
    print(f"{agency}: {percentage:.2f}%")
```

```
Total spending for 2023: $7 882 800 000 000.00
Percentages of the total spending for 2023:
Department of Defense: 12.73%
Department of Veterans Affairs: 18.46%
Interest on Treasury Securities held by the public: 8.60%
National Science Foundation: 0.11%
Department of Transportation: 1.35%
Department of Education: 0.23%
Department of Energy: 0.95%
```

2.10 PRICES OF PRECIOUS METALS

In this study, we will also see the price movement of precious metals. They have proven themselves over the millennia as "independent money", materialized labor. They cannot be falsified in the general case, they cannot be printed, they cannot be created with a few clicks on a computer keyboard. When they are mined, it is not according to cleverly invented algorithms, but actually from the ground to the finished product they travel a long way, which involves a lot of work and energy. Here we will look at 2 precious metals:

- **Gold** as a symbol of millennial **world money**.
- **Platinum** as world money, but also as an **industrial metal**.

With these metals we will also compare the economic indicators of the USA, because of their quality as real money and raw material for many industrial productions. From this site investing.com we only download information for the period of our study: **XAU/USD - Gold Spot US Dollar**. [11] That is, we have the price of 24 karat gold per troy ounce in dollars. From the same site, we also download the monthly platinum prices for the time interval we are looking for: **XPT/USD Historical Data** [12]

```
# Loading GOLD data for 2022 and 2024.
xau = pd.read_csv('data/XAU_USD_Historical_Data.csv')
# We leave only the requested columns...
keep_col = ['Date', 'Price']
xau_filtered = xau[keep_col]
del xau
# Rename columns.
xau_filtered.columns = ['date', 'xau_price']

# We remove the commas and convert to numeric type.
xau_filtered['xau_price'] = xau_filtered['xau_price'].str.replace(',', '').asty
pe(float)
# We show the first few lines for verification!
xau_filtered.head()
```

	date	xau_price
0	06/01/2024	2324.98
1	05/01/2024	2326.97
2	04/01/2024	2284.57

```
3  03/01/2024    2232.38
4  02/01/2024    2043.24
```

```
xau_filtered[['xau_price']].describe().T # GOLD
```

```
           count         mean         std      min        25%       50%  \
xau_price   24.0  1959.966667  193.751874  1633.12  1826.4625  1963.245

                75%      max
xau_price  2038.7025  2326.97
```

```
# Loading PLATINUM data for 2022 and 2024.
xpt = pd.read_csv('data/XPT_USD_Historical_Data.csv')
# We leave only the requested columns...
keep_col = ['Date', 'Price']
xpt_filtered = xpt[keep_col]
del xpt
# Rename columns.
xpt_filtered.columns = ['date', 'xpt_price']

# We remove the commas and convert to numeric type.
xpt_filtered['xpt_price'] = xpt_filtered['xpt_price'].str.replace(',', '').asty
pe(float)
# We show the first few lines for verification.
xpt_filtered.head()
```

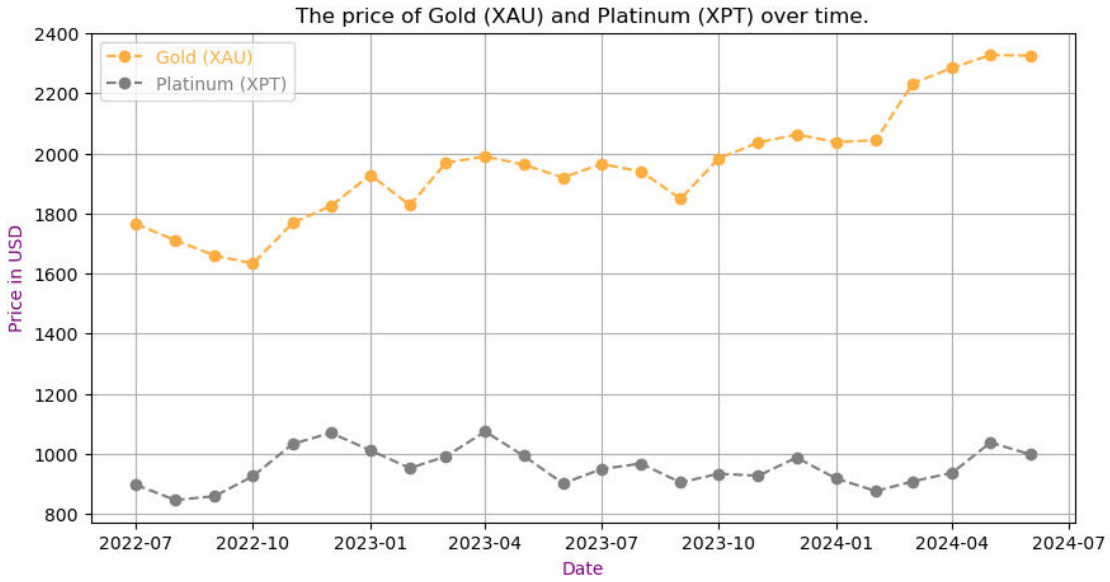
```
         date  xpt_price
0  06/01/2024     998.00
1  05/01/2024    1037.70
2  04/01/2024     936.75
3  03/01/2024     908.05
4  02/01/2024     875.80
```

```
xpt_filtered[['xpt_price']].describe().T # PLATINUM
```

```
           count        mean        std     min       25%      50%       75%  \
xpt_price   24.0  954.030833  63.013305  846.15  907.1525  942.975  994.4375

              max
xpt_price  1074.0
```

```
# We convert the 'date' columns to a datetime type.
xau_filtered['date'] = pd.to_datetime(xau_filtered['date'])
xpt_filtered['date'] = pd.to_datetime(xpt_filtered['date'])
# We sort the data by date.
xau_filtered = xau_filtered.sort_values('date')
xpt_filtered = xpt_filtered.sort_values('date')

# Visualization of prices over time.
plt.figure(figsize=(10, 5))
plt.plot(xau_filtered['date'], xau_filtered['xau_price'], marker='o', linestyle
='--', color = '#FFAE42')
plt.plot(xpt_filtered['date'], xpt_filtered['xpt_price'], marker='o', linestyle
='--', color = 'grey')
```

```
plt.title('The price of Gold (XAU) and Platinum (XPT) over time.')
plt.xlabel('Date', color = 'purple')
plt.ylabel('Price in USD', color = 'purple')

legend = plt.legend(['Gold (XAU)', 'Platinum (XPT)'], loc='upper left')
legend.texts[0].set_color('#FFAE42')
legend.texts[1].set_color('grey')

plt.grid(True)
plt.show()
```



2.11 Gathering the data into a single dataframe.

Collecting **only all time series from July 2022 to June 2024** in one table. Percentages are converted to numbers and only data with a dollar value is taken. Not in millions or billions.

```
# Create a new empty table.
collected_data = pd.DataFrame(columns=[
    'cci',
    'cpi',
    'fed_funds',
    'unemployment',
    'trade_balance',
    'total_debt',
    'gold_price',
    'platinum_price',
    'gdp_approx'
])
# Add data to the columns.
collected_data['cci'] = filtered_cci['cci']
collected_data['cpi'] = cpi['cpi']
collected_data['fed_funds'] = fed_funds['fed_funds'] / 100 # From % to numbers.
collected_data['trade_balance'] = tb['trade_balance']
collected_data['total_debt'] = debt_real['total_debt']
collected_data['unemployment'] = unemployment['unemployment'] / 100 # % to num.
collected_data['gold_price'] = xau_filtered['xau_price']
collected_data['platinum_price'] = xpt_filtered['xpt_price']
collected_data['gdp_approx'] = gdp_approx['gdp']
```

```
collected_data
# Show all data: _______ DataFrame (DF) with real data! _______
```

	cci	cpi	fed_funds	unemployment	trade_balance	total_debt	\
0	95.3	294.977	0.0168	0.035	-8.923600e+10	3.059511e+13	
1	103.6	295.209	0.0233	0.036	-8.678200e+10	3.093608e+13	
2	107.8	296.341	0.0256	0.035	-9.001500e+10	3.092891e+13	
3	102.5	297.863	0.0308	0.036	-9.614500e+10	3.123830e+13	
4	101.4	298.648	0.0378	0.036	-8.397500e+10	3.141332e+13	
5	108.3	298.812	0.0410	0.035	-9.271100e+10	3.141969e+13	
6	106.0	300.356	0.0433	0.034	-9.125800e+10	3.145498e+13	
7	103.4	301.509	0.0457	0.036	-9.244400e+10	3.145929e+13	
8	104.0	301.744	0.0465	0.035	-8.314900e+10	3.145844e+13	
9	101.3	303.032	0.0483	0.034	-9.576600e+10	3.145782e+13	
10	102.5	303.365	0.0506	0.037	-8.980300e+10	3.146446e+13	
11	110.1	304.003	0.0508	0.036	-8.842300e+10	3.233227e+13	
12	114.0	304.628	0.0512	0.035	-8.841400e+10	3.260859e+13	
13	108.7	306.187	0.0533	0.038	-8.370000e+10	3.291415e+13	
14	104.3	307.288	0.0533	0.038	-8.571500e+10	3.316733e+13	
15	99.1	307.531	0.0533	0.038	-8.782900e+10	3.369958e+13	
16	101.0	308.024	0.0533	0.037	-8.834800e+10	3.387868e+13	
17	110.7	308.742	0.0533	0.037	-8.844000e+10	3.400149e+13	
18	110.9	309.685	0.0533	0.037	-9.119600e+10	3.419115e+13	
19	104.8	311.054	0.0533	0.039	-9.318300e+10	3.447108e+13	
20	103.1	312.230	0.0533	0.038	-9.334700e+10	3.458653e+13	
21	97.5	313.207	0.0533	0.039	-9.927700e+10	3.461699e+13	
22	102.0	313.225	0.0533	0.040	-9.984300e+10	3.466712e+13	
23	97.8	313.049	0.0533	0.041	-9.735200e+10	3.483163e+13	

	gold_price	platinum_price	gdp_approx
0	2324.98	998.00	2.599464e+13
1	2326.97	1037.70	2.613256e+13
2	2284.57	936.75	2.627048e+13
3	2232.38	908.05	2.640840e+13
4	2043.24	875.80	2.654347e+13
5	2037.19	917.91	2.667854e+13
6	2062.59	987.25	2.681360e+13
7	2035.75	926.93	2.689674e+13
8	1983.01	933.67	2.697988e+13
9	1848.58	904.46	2.706301e+13
10	1939.74	967.54	2.724538e+13
11	1964.19	949.20	2.742776e+13
12	1919.57	901.25	2.761013e+13
13	1962.30	993.25	2.772575e+13
14	1989.65	1074.00	2.784137e+13
15	1967.90	991.31	2.795700e+13
16	1827.15	952.35	2.806106e+13
17	1927.88	1011.15	2.816512e+13
18	1824.40	1069.58	2.826917e+13
19	1768.45	1032.72	2.838917e+13
20	1633.12	925.72	2.850916e+13
21	1659.67	859.00	2.862915e+13

```
22      1710.70          846.15  2.862915e+13
23      1765.22          897.00  2.862915e+13
```

3. Analysis and Hypothesys testing

3.1 Stationarity

Autocorrelation measures the dependence of time series on its own past values. Examining autocorrelation is important for time series because it can reveal patterns, seasonality, and dependence in the data. In Python, we can use the statsmodels library to calculate and visualize the autocorrelation. Before analyzing autocorrelation, **it is useful to check that the time series is stationary**. We can use the Dickey-Fuller (ADF) test for this purpose. The code below will help us examine the autocorrelation of our time series and check if they are stationary, which is important for proper time series analysis and modeling. Comparing the graphs of the data, we choose **the most monotonic function: CPI**.

Is there a statistically significant difference in the Consumer Price Index (CPI) over the time period under our study?

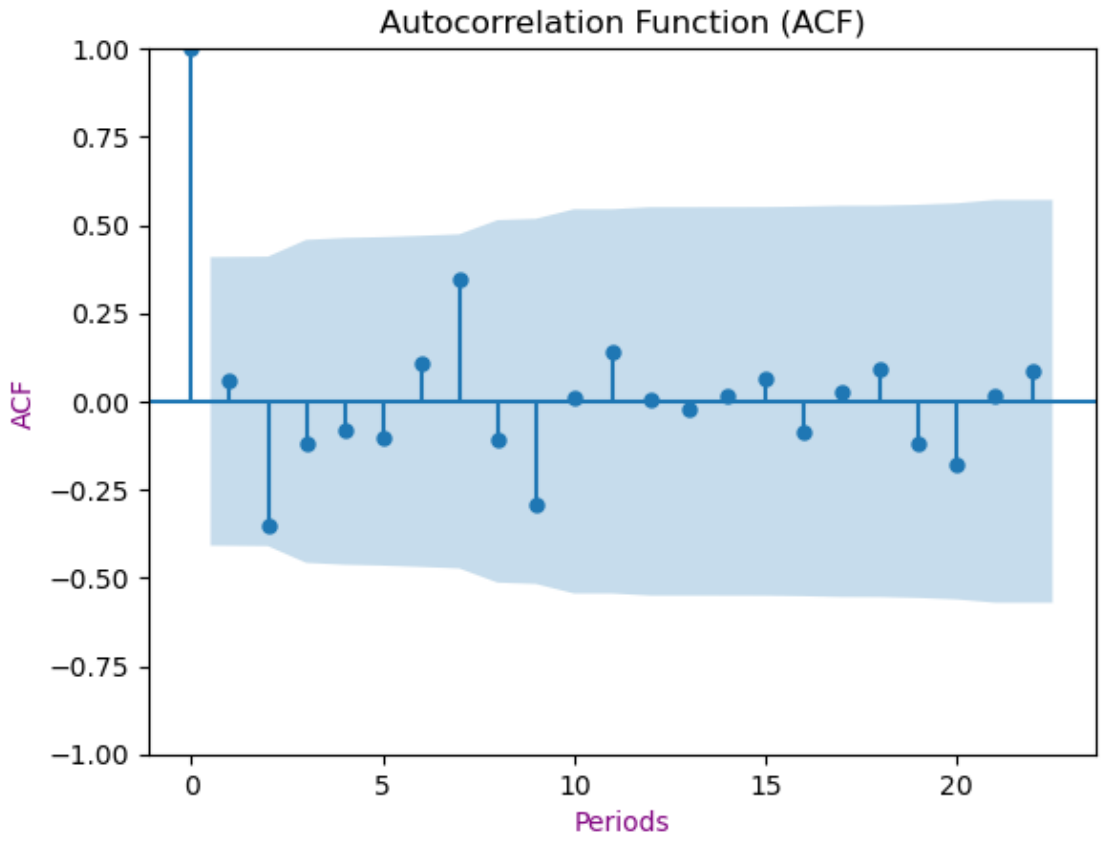
- H_0 : The **data has a unit root and is non-stationary**
  (**p-value > 0.05**: Fail to reject the null hypothesis (H0)).
- H_1 : The **data does not have a unit root and is stationary**
  (**p-value <= 0.05**: Reject the null hypothesis (H0)).
- **alpha = 0.05**

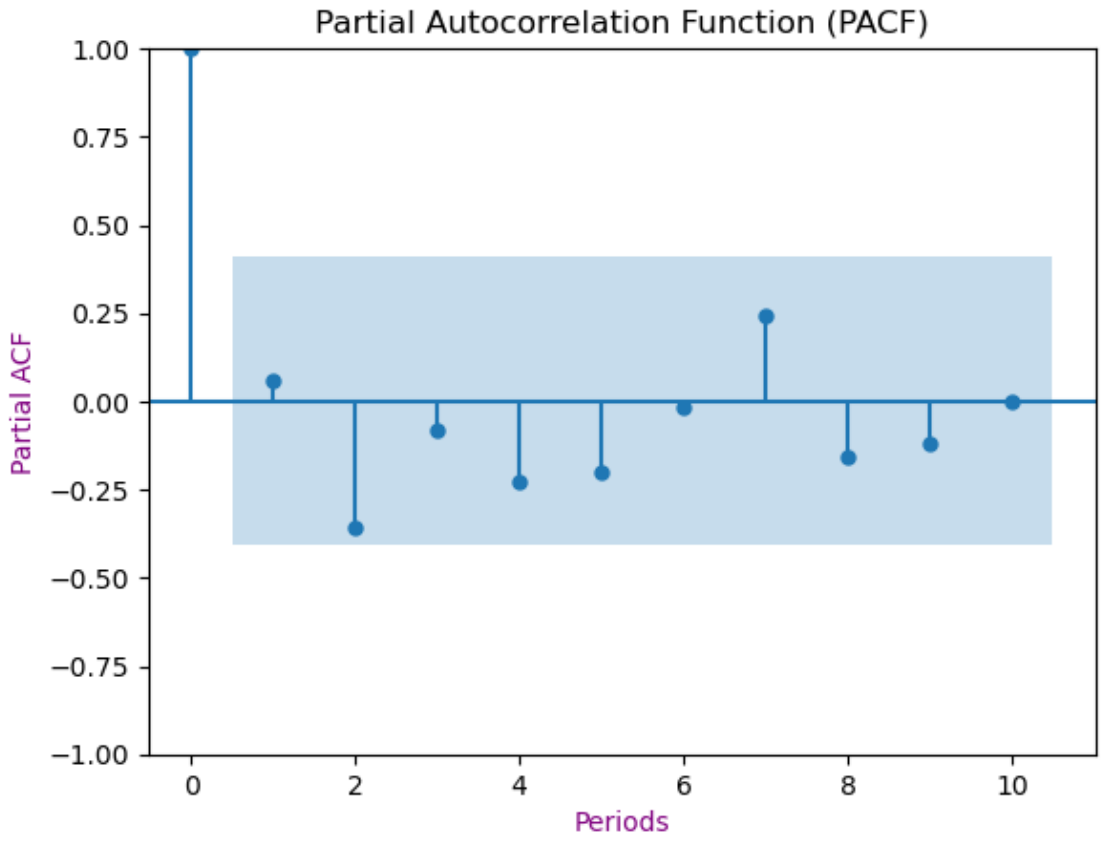
Rejecting the null hypothesis means that the process has no unit root, and in turn that the time series is stationary or does not have time-dependent structure. If the time series is stationary, continue to the next steps. If the time series is not stationary, try differencing the time series and check its stationarity again. Time Series: Interpreting ACF and PACF [13]

```
# We select the column we want to analyze (eg 'cci').
time_series = collected_data['cpi'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stationary, we can make it stationary by differenti
ation with: time_series.diff().dropna()

# If the p-value is greater than 0.05, the series is not stationary.
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
```

```
# We want to see the autocorrelation for up to 22 lag periods.
plot_acf(time_series, lags=22)
plt.title('Autocorrelation Function (ACF)')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()

# Partial autocorrelation test (PACF)
plt.figure(figsize=(10, 6))

# The partial autocorrelation here works for up to 10 lag periods.
plot_pacf(time_series, lags=10)
plt.title('Partial Autocorrelation Function (PACF)')
plt.xlabel("Periods", color = "purple")
plt.ylabel("Partial ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -1.2192149080973167
p-value: 0.6652758105336317
Critical Value (1%): -3.889265672705068
Critical Value (5%): -3.0543579727254224
Critical Value (10%): -2.66698384083045

Fail to reject the null hypothesis (H0),
the data has a unit root and is non-stationary.
```





The **p-value** is much greater than 0.05, the line is **not stationary**, therefore the other **time series share the same or similar characteristics**. Interpretation:

- **Positive ACF**: A positive ACF at lag k indicates a positive correlation between the current observation and the observation at lag k.
- **Negative ACF**: A negative ACF at lag k indicates a negative correlation between the current observation and the observation at lag k.
- **Decay in ACF**: The decay in autocorrelation as lag increases often signifies the presence of a trend or seasonality in the time series.
- **Significance**: Significant ACF values at certain lags may suggest potential patterns or relationships in the time series. Autocorrelation and Partial Autocorrelation [14]

The analysis of ACF and PACF will be done in the Conclusions section of the present study.

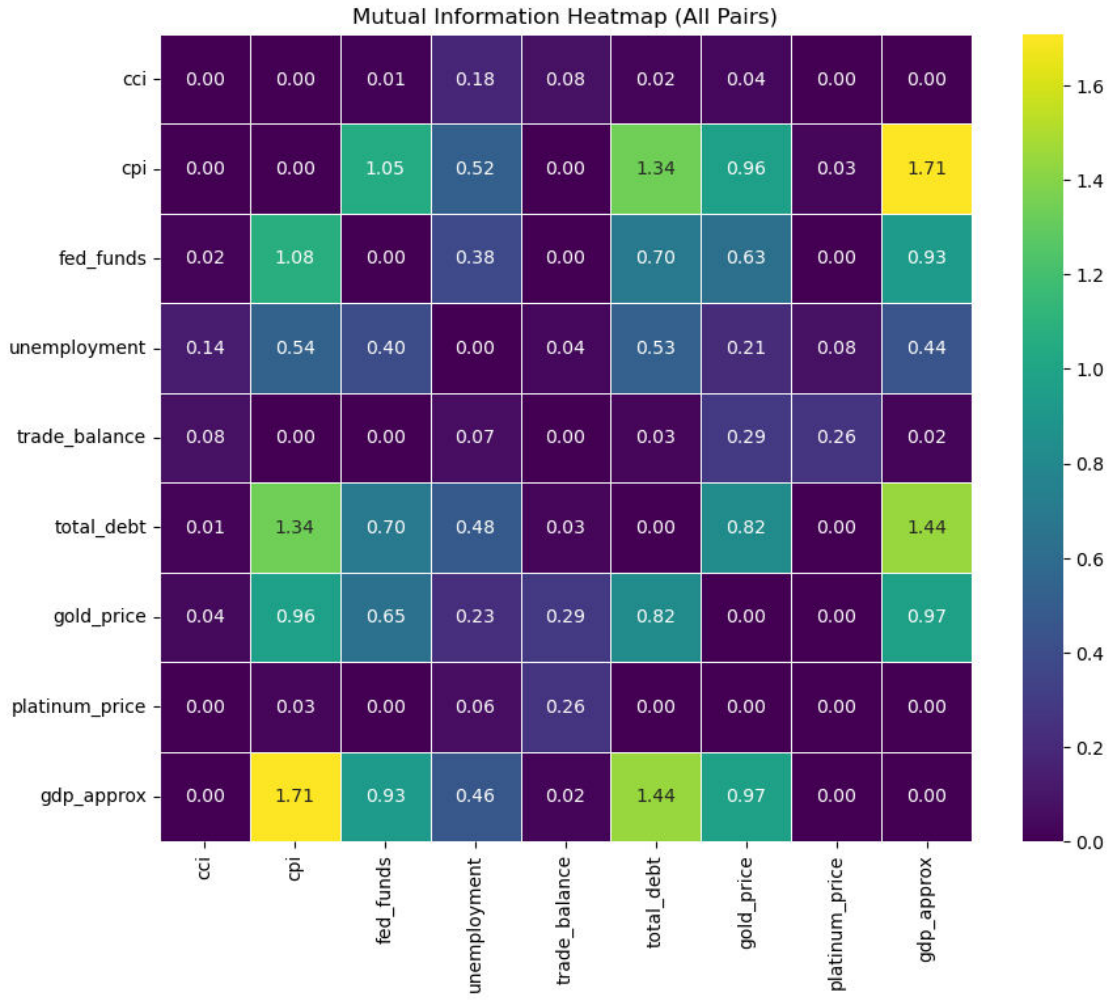
For **better fundamental analysis, a combined approach involving several different methods for detecting dependencies** is often used. For example:

1. Using **mutual information** to initially detect **non-linear dependencies**.
2. **Verification** of results with **Spearman** and **Kendall** correlations.
3. **Regression analysis for deeper understanding** of relationships.

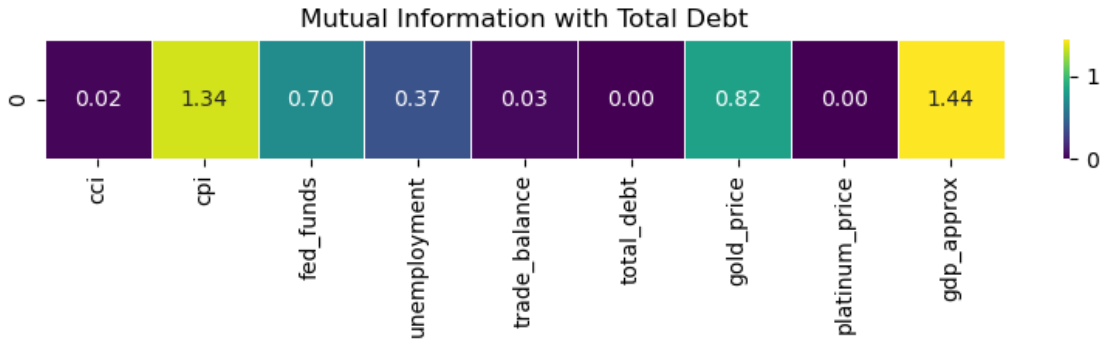
3.2 Using mutual information to initially detect non-linear dependencies.

To study non-linear dependencies, we can use mutual information (**Mutual Information**), which measures dependencies between variables without being limited to linear relationships. Mutual Information is a measure that assesses the dependence between two variables. **It is not exactly a correlation, but rather a measure of how much information the two variables share**. This means that mutual information measures how much knowledge about one variable gives knowledge about another variable. In this context, mutual information between two variables **can detect non-linear relationships** that classical linear correlations (such as Pearson's) cannot detect. To visualize the results, we can use a heatmap.

```
# Function for calculating the mutual information (mutual information) between
columns in a DataFrame.
def calculate_mutual_info(df):
# We create an empty DataFrame with an index and columns equal to the column
names of the input DataFrame.
    mi_matrix = pd.DataFrame(index=df.columns, columns=df.columns)

    # An outer loop that loops through each column (col1) in the DataFrame.
    for col1 in df.columns:
        # An inner loop that loops through each column (col2) in the DataFrame.
        for col2 in df.columns:
            # We check if column col1 is not equal to column col2.
            if col1 != col2:
                # We calculate the mutual information between col1 and col2.
                mi_col1_col2 = mutual_info_regression(df[[col1]], df[col2])[0]

                # We calculate the mutual information between col2 and col1.
                mi_col2_col1 = mutual_info_regression(df[[col2]], df[col1])[0]

     # Calculate the average of the 2 measurements and record it in the matrix.
                mi_matrix.loc[col1, col2] = (mi_col1_col2 + mi_col2_col1) / 2
            else:
                # We set the diagonal elements (where col1 equals col2) to 0.
                # This is optional, but usually the mutual information between
                # a column is itself zero or insignificant.
                mi_matrix.loc[col1, col2] = 0

    # We return the matrix of mutual information with data type float.
    return mi_matrix.astype(float)

# Calculate the mutual information between all columns.
mi_matrix = calculate_mutual_info(collected_data)

# Uses the seaborn library to create a heatmap. Each cell in the
# heatmap shows the mutual information value between the corresponding columns.

# We make sure we close all previous shapes before creating the new heatmap.
plt.close('all')

# Visualization of mutual information with a heat map.
plt.figure(figsize=(10, 8))

# Add border lines for better visibility.
sns.heatmap(mi_matrix, annot=True, cmap='viridis', fmt='.2f', linewidths=0.5)

# Customize title.
plt.title('Mutual Information Heatmap (All Pairs)')
plt.show()
```




```
# Calculate the mutual information between each column and total_debt.
mi_debt = pd.Series(index=collected_data.columns)
for col in collected_data.columns:
    if col != 'total_debt':
        mi = mutual_info_regression(collected_data[[col]], collected_data['tota
l_debt'])[0]
        mi_debt[col] = mi
    else:
        mi_debt[col] = 0

# Visualization of mutual information.
plt.figure(figsize=(10, 1))
sns.heatmap(mi_debt.to_frame().T, annot=True, fmt='.2f', cmap='viridis', cbar=T
rue, linewidths=0.5)
plt.title('Mutual Information with Total Debt')
plt.show()
```



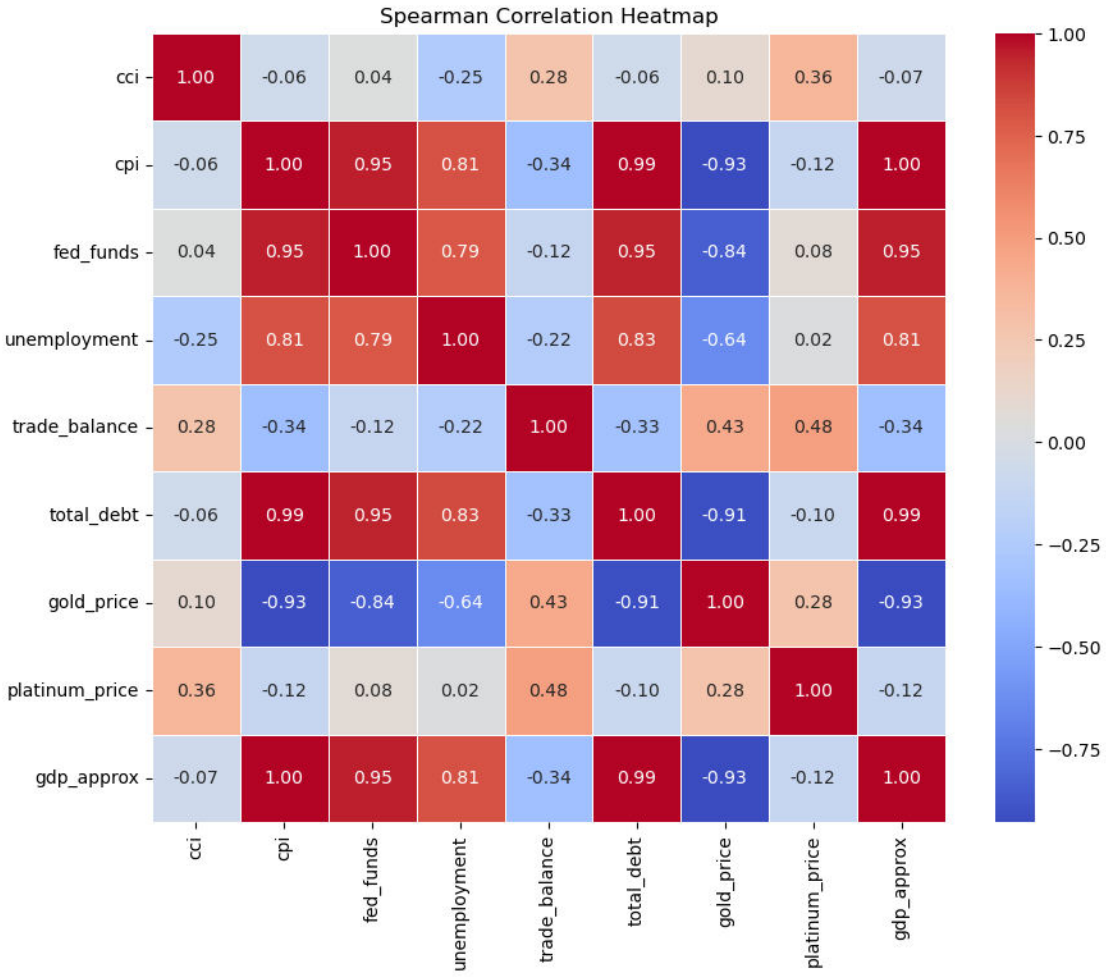
Here are the highlights of mutual information:

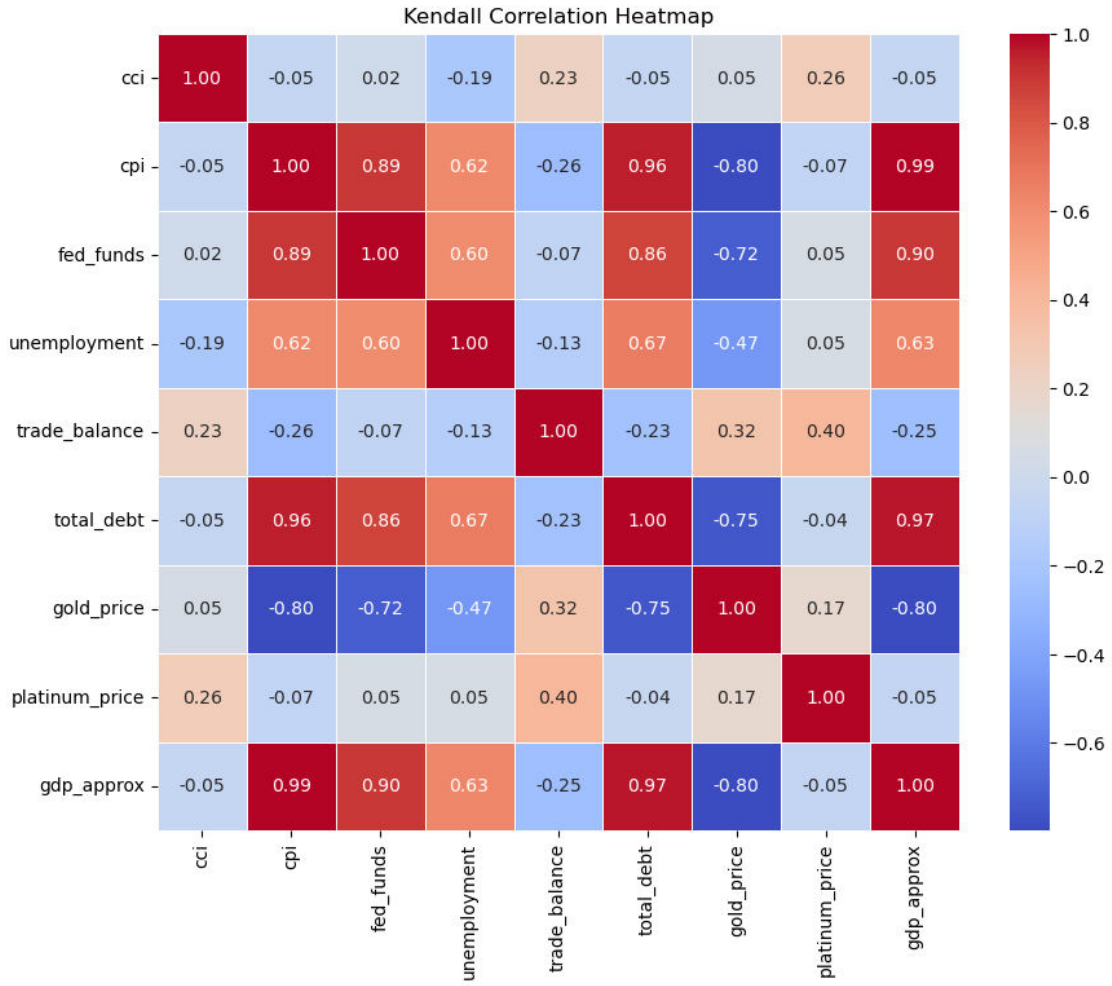
- **Values**: Mutual information is always non-negative. A value of 0 means that the two variables are completely independent. Larger values indicate stronger dependence, but there is no upper limit.
- **Interpretation**: Higher mutual information values mean greater dependence between variables. A value of **1.34**, for example, indicates that there is a significant relationship between **Total Debt** and **CPI**, although the exact interpretation of the value is not as straightforward as with correlation.
- The analysis of Mutual Information will be given in the Conclusions section of the present study.

3.3 Verification of results with Spearman and Kendall correlations.

- **Spearman** Correlation: Used to measure monotonic dependencies (can detect non-linear monotonic relationships).
- **Kendall's** Correlation: Estimates linear relationships.
- The analysis of this correlations will be given in the Conclusions section of the present study.

```
# Calculation of Spearman and Kendall correlations.
spearman_corr = collected_data.corr(method='spearman')
kendall_corr = collected_data.corr(method='kendall')

# Visualization of Spearman correlations.
plt.figure(figsize=(10, 8))
sns.heatmap(spearman_corr, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0
.5)
plt.title('Spearman Correlation Heatmap')
plt.show()

# Visualization of Kendall correlations.
plt.figure(figsize=(10, 8))
sns.heatmap(kendall_corr, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.
5)
plt.title('Kendall Correlation Heatmap')
plt.show()
```


Spearman Correlation Heatmap
cci
cpi
fed_funds
unemployment
trade_balance
total_debt
gold_price
platinum_price
gdp_approx
1.00 -0.06 0.04 -0.25 0.28 -0.06 0.10 0.36 -0.07
-0.06 1.00 0.95 0.81 -0.34 0.99 -0.93 -0.12 1.00
0.04 0.95 1.00 0.79 -0.12 0.95 -0.84 0.08 0.95
-0.25 0.81 0.79 1.00 -0.22 0.83 -0.64 0.02 0.81
0.28 -0.34 -0.12 -0.22 1.00 -0.33 0.43 0.48 -0.34
-0.06 0.99 0.95 0.83 -0.33 1.00 -0.91 -0.10 0.99
0.10 -0.93 -0.84 -0.64 0.43 -0.91 1.00 0.28 -0.93
0.36 -0.12 0.08 0.02 0.48 -0.10 0.28 1.00 -0.12
-0.07 1.00 0.95 0.81 -0.34 0.99 -0.93 -0.12 1.00
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75



3.4 Regression analysis for deeper understanding of relationships.

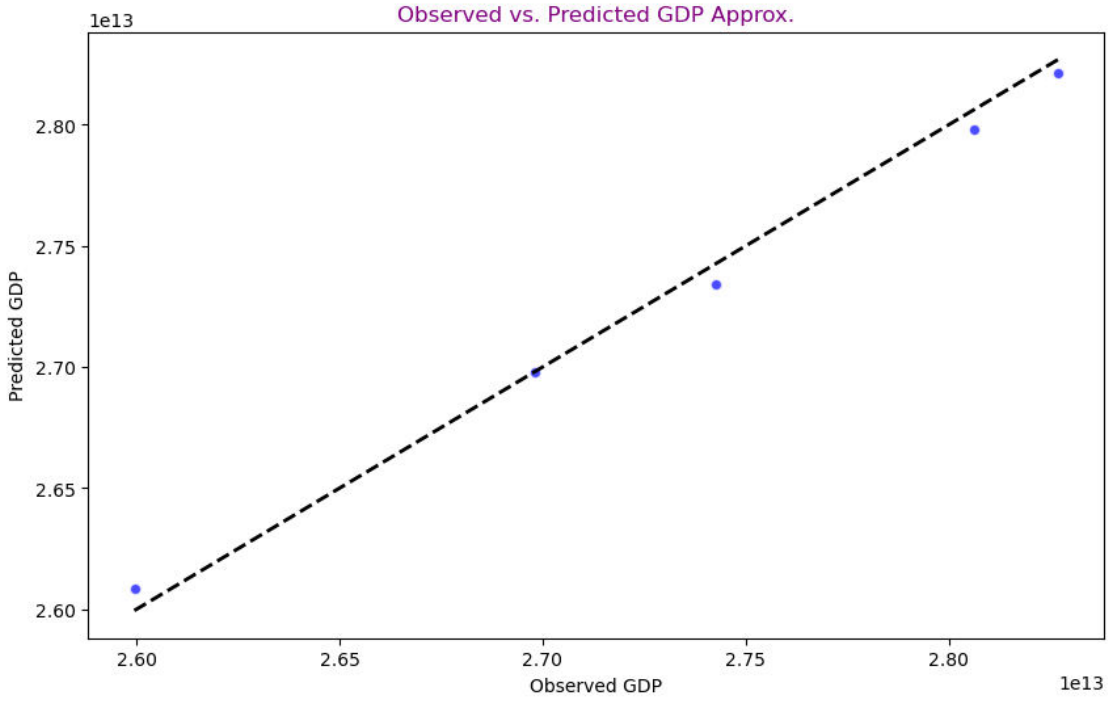
Let's use **linear regression** to analyze the **relationship** between **gdp_approx**, **cpi** and **total_debt**. We assume that the relationship between the independent variables (in this case cpi and total_debt) and the dependent variable (gdp_approx) is linear. This means that the changes in the dependent variable are proportional to the changes in the independent variables.

Based on the proven strong relationship between CPI, DEBT and GDP, we can build a **model that predicts GDP** using **CPI** and **DEBT** as input variables.

```
# Data preparation.
data_frame = pd.DataFrame(collected_data)

# Select the independent variables (features) and the dependent (target).
X = data_frame[['cpi', 'total_debt']]
y = data_frame['gdp_approx']

# We split the data into a training set and a test set.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, rand
om_state = 42)

# We build and train the linear regression model.
model = LinearRegression()
# Training the model.
model.fit(X_train, y_train)

# We predict the values of the test data.
```

```
y_pred = model.predict(X_test)

# We calculate the evaluation metrics.
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# We display the results.
print(f'Mean Squared Error: {mse}')
print(f'R-squared: {r2}')

# Preview the results.
plt.figure(figsize = (10, 6))
plt.scatter(y_test, y_pred, color = 'blue', edgecolor = 'w', alpha = 0.7)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw
= 2)
plt.xlabel('Observed GDP')
plt.ylabel('Predicted GDP')
plt.title('Observed vs. Predicted GDP Approx.', color = 'purple')
plt.show()
```

```
Mean Squared Error: 5.015412256027562e+21
R-squared: 0.9924703981731748
```



Mean Squared Error (MSE)

- MSE = 5.015412256027562e+21 MSE measures the mean of the squared errors, where the errors are the differences between the predicted values and the actual values. A lower MSE value indicates that the model is more accurate. In our case, the MSE is quite large, which is expected given the scale of the data (trillions for gdp_approx).

  R-squared ($R^2 = 1 - \frac{Unexplained\ Variation}{Total\ Variation}$ **)**

- R-squared = 0.9924703981731748 R-squared shows how much of the variation in the dependent variable (gdp_approx) can be explained by the independent variables (cpi and total_debt). The R-squared value is between 0 and 1:
  - A value close to 1 means that the model explains a very large proportion of the variation.
  - A value close to 0 means that the model does not explain the variation well.

In our case, the R-squared is 0.9924703981731748, which is very close to 1. This indicates that the **model explains 99.24% of the variation in GDP based on the CPI and DEBT** variables. This is a very high score and indicates that the model has very good predictive ability.

The modeling analysis will be given in the section below.

4. Conclusions

1. Significant financial upheaval is potentially ahead of the financial world as **Saudi Arabia has decided not to renew its 50-year petro-dollar deal** (originally signed on 8 June 1974) with the United States. The deal, which expired on Sunday 9 June, was a cornerstone of the United States global economic dominance, The Business Standard commented. [15] With the disappearance of the petro-dollar, it will be difficult to export inflation outside the US. But this fact will be reflected already outside the framework of our studied time interval.
2. When we look at the general graph of gold and platinum, we notice that platinum, being more of an industrial metal, has a price that does not fluctuate much over time. While gold, especially after September 2023, has a steady trend of price increase. This can lead us to think that **there is inflationary expectations in entire the world**, which can be confirmed by many articles like this one: Gold Shines So Far in 2024 as Central Banks Invest Heavily. [16]
3. From the US spending survey for 2022 and 2023, we see the **five heaviest feathers**:
   - Department of Veterans Affairs. (**18.46%** for 2023)
   - Department of Health and Human Services.
   - Social Security Administration.
   - Department of Defense. (**12.73%** for 2023)
   - Interest on Treasury Securities held by the public. (**8.60%** for 2023)

From the graphs depicting the US budget, we see that **these 5 categories are far ahead of all others**. Of these, only two (the Department of Health and Human Services and the Social Security Administration) contribute to the welfare of the USA. Much of the spending is on wars and their consequences (Department of Veterans Affairs). A significant part of the budget also goes to pay interest on government loans. **The feathers of the budget that make a competitive economy with good infrastructure are poorly funded or with very symbolic funding**. Such as:

- Small Business Administration.
- National Science Foundation. (**0.11%** for 2023)
- Environmental Protection Agency.
- Department of Transportation. (**1.35%** for 2023)
- Department of Energy. (**0.95%** for 2023)
- Department of Education. (**0.23%** for 2023)
- Department of Agriculture.

Looking at the distribution of the funds of the US budget (which is mainly filled not by production, but by services), we does not see good prospects in the future. It is not invested in the development of the USA, but in wars, their consequences and interest on the loans taken.

4. The analysis of **ACF** and **PACF**:

- The ACF chart is too short to see any discernible trend. It looks like a random distribution. There is some decay and we can assume that there is some weak trend.
- The PACF graph is even shorter and definitely cannot be deduced from it or find any pattern of behavior.

5. The analysis of **Mutual Information**:
   - From the Mutual Information Heatmap, we see that there is a strong dependence between **Total Debt** and **CPI**,**Total Debt** and **GDP**, and especially between **GDP** and **CPI**.
   - Mutual Information is a measure that assesses the dependence between two variables. Rather a measure ofhow much information the two variables share.
   - From this we can conclude that the **growth of the US economy is mainly affected by inflation and the accumulation of more debt**.
6. The analysis of **Spearman and Kendall correlations**:

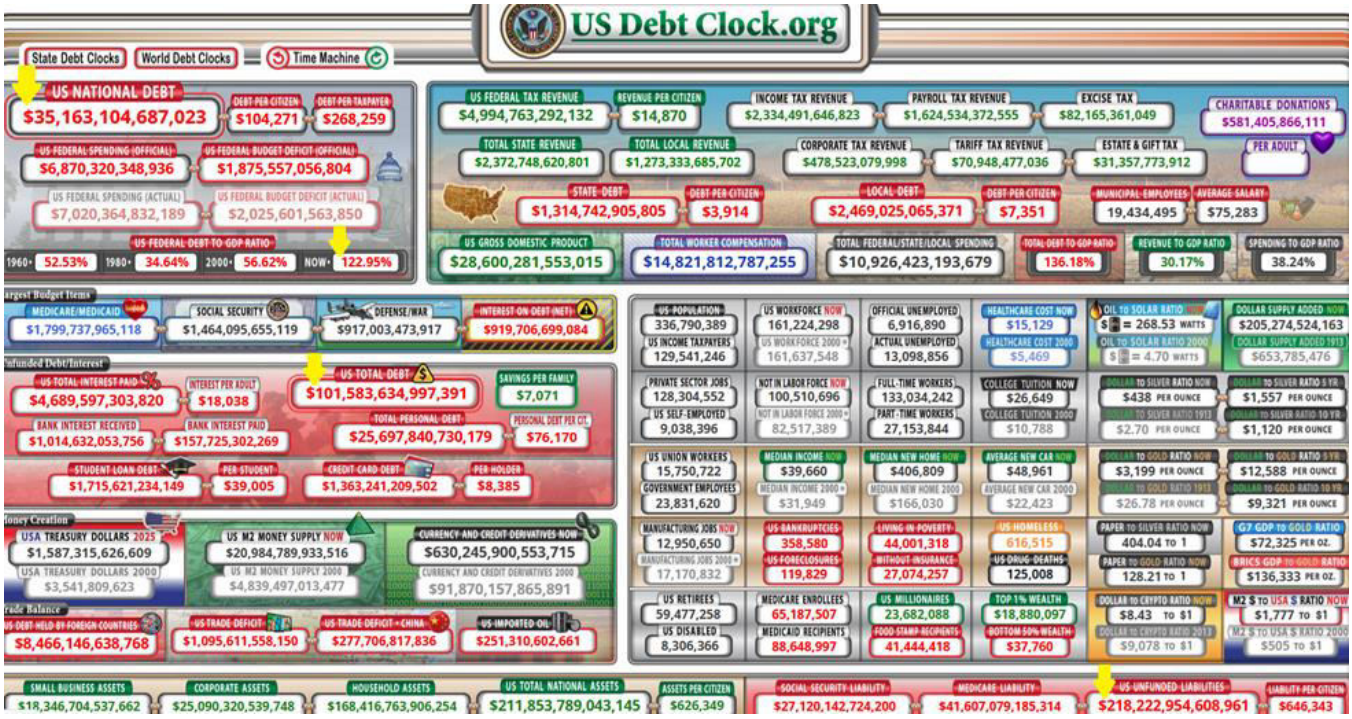
   Let us first consider **Kendall's** linear correlation dependences:
   - The price of **GOLD** has a negative correlation with all the important categories listed below.
   - **CPI** has a strong positive correlation with FED FUNDS, DEBT and especially with GDP (+0.99).
   - **UNEMPLOYMENT** has a positive correlation with DEBT.
   - **DEBT** has a strong positive correlation with FED FUNDS, CPI and GDP.
   - **GDP** has a strong positive correlation with DEBT, CPI and FED FUNDS.

   Now let's look at non-linear **Spearman** correlations:
   - Here we will **only** mention the **differences with Kendall** ie. where there arestill nonlinear dependencies found.
   - The price of **GOLD** has again a negative correlation with all the important categories listed below.We would explain this by the fact that gold is "independent money" and does not fitinto the "debt vs. development" game.
   - All other correlations noted in Kendall **here only deepen their correlation**. This is easily tracked through the coefficients.

   This again confirms the last thesis from item 5:

   Growth of the US economy is mainly affected by **inflation** and the accumulation of **more debt**.
7. The visualization of the **regression analysis** gives a real insight into the accuracy of the model. R-squared = 0.99247 is a very high score and indicates that the model has very good predictive ability. The selected linear regression model is very good for **predicting GDP based on CPI and DEBT**. This again proves their very close dependence.

8. The main weight of the US budget mainly goes to military purposes and the consequences of wars. The share of interest payments on borrowed loans is also very large. The almost direct linear dependence of the expansion of the economy on inflation (printing money) and borrowing was also shown. The latter, since the dollar is a world currency, means printing money again. As a result of all that has been said so far, we can safely summarize that **the American economy is addicted to printing new money.** The trade balance is strongly negative with a worsening trend. Science is not a priority. **If for some reason the printing of new money stops** and/or the dollars of the world go "home" to the USA, **we are waiting for "Great Depression - 2"**. How serious is the problem? Let's see it in a picture from the US Debt Clock website. [17] The date is: **16.08.2024 18:30**



US FEDERAL DEBT TO GDP RATIO = **122.95** % (Source: FEDERAL RESERVE). No!

US NATIONAL DEBT = **35 163 105 000 000** USD (Source: US TREASURY). Uh-uh, no.

US TOTAL DEBT = **101 583 635 000 000** USD (Source: FEDERAL RESERVE). Maybe...

US UNFUNDED LIABILITIES = **218 222 955 000 000** USD (Source: US TREASURY). Cheers!

Resources:

1. **Importance of Economic Indicators**: https://www.fxcc.com/eu/major-economic-indicators
2. **statista - Global inflation rate from 2000 to 2022**: https://www.statista.com/statistics/256598/global-inflation-rate-compared-to-previous-year/
3. **FEDERAL RESERVE BANK of ST. LOUIS**: https://fred.stlouisfed.org/series/GDP
4. **FEDERAL RESERVE BANK of ST. LOUIS**: https://fred.stlouisfed.org/series/CPIAUCSL
5. **United States Conference Board Consumer Confidence**: https://macrovar.com/united-states/consumer-confidence-index/
6. **FEDERAL RESERVE BANK of ST. LOUIS**: https://fred.stlouisfed.org/series/FEDFUNDS
7. **FEDERAL RESERVE BANK of ST. LOUIS**: https://fred.stlouisfed.org/series/BOPGTB
8. **FiscalData**: https://fiscaldata.treasury.gov/datasets/monthly-statement-public-debt/summary-of-treasury-securities-outstanding
9. **FEDERAL RESERVE BANK of ST. LOUIS**: https://fred.stlouisfed.org/series/LRHUTTTTUSM156S
10. **FiscalData**: https://fiscaldata.treasury.gov/datasets/u-s-government-financial-report/statements-of-net-cost)
11. **investing.com**: https://www.investing.com/currencies/xau-usd-historical-data
12. **investing.com**: https://www.investing.com/currencies/xpt-usd-historical-data
13. **Time Series: Interpreting ACF and PACF**: https://www.kaggle.com/code/iamleonie/time-series-interpreting-acf-and-pacf/notebook?scriptVersionId=80091228
14. **Autocorrelation and Partial Autocorrelation**: https://www.geeksforgeeks.org/autocorrelation-and-partial-autocorrelation/

15. **Business Standard - Saudi Arabia's petro-dollar exit**: https://www.tbsnews.net/world/global-economy/saudi-arabias-petro-dollar-exit-global-finance-paradigm-shift-875321
16. **Gold Shines So Far in 2024 as Central Banks Invest Heavily**: https://www.investopedia.com/gold-shines-so-far-in-2024-as-central-banks-invest-heavily-8672711
17. **US Debt Clock**: https://usdebtclock.org/

Chapter 3: **Analysis of the US Economy – part 2**

```
# import libraries
import pandas as pd
import numpy as np
import seaborn as sns
import plotly.express as px
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from datetime import datetime, timedelta
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import TimeSeriesSplit
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
from catboost import CatBoostClassifier
from sklearn.metrics import (
    roc_curve, auc, confusion_matrix, f1_score, precision_score,
    recall_score, accuracy_score, roc_auc_score, classification_report )
from sklearn.preprocessing import StandardScaler
from matplotlib.dates import YearLocator, DateFormatter
```

Date: November 2024, Machine Learning project

Abstract:

This study, based on statistical indicators of the US economy, attempts to predict the probability of a recession in the US economy. Three methods of prediction are shown, and the corresponding conclusions from the study are drawn...

1. Defining the problem and main assumptions

In the **first part** of this study, looking at the distribution of funds from the US budget, we did not see good prospects in the future. It is not invested in the development of the USA, but in the wars, their consequences and the interest on the loans taken. The share of interest payments on borrowed loans is also very large. The almost direct linear dependence of the expansion of the economy on inflation (printing money) and borrowing was also shown. The latter, since the dollar is a world currency, means printing money again. As a result of all that has been said so far, we can safely summarize that the American economy is addicted to printing new money. The trade balance is strongly negative with a worsening trend. Science is not a priority. If for some reason the printing of new money stops and/or the dollars of the world go "home" to the USA, we are waiting for "Great Depression - 2".

From here, we can ask ourselves the following question: **Can we use ML to predict approximately when the US economy will go into recession?** The **second part** of the present study will be based on the answer to this question.

There are **2 main theoretical assumptions** at play:

- U.S. recessions exhibit markers / early warning signs. There exist plenty of recession “signals” in the form of individual economic or market data series. While individually, these signals have limited information value, they may become more useful when combined together.

- Future recessions will be similar to historical recessions. This assumption is a lot more shaky, but can be mitigated by choosing features that maintain significance despite the changing economic landscape. For example, focusing on manufacturing data may have been relevant historically, but may be less relevant going forward as the world goes digital.

What Have Others Tried?

- **Guggenheim Partners** has 2 recession related indicators: a Recession Probability Model and a Recession Dashboard, both driven by a combination of economic and market indicators. They try to predict recession probabilities across 3 different time frames.
- **New York FED** predicts recession probabilities. Its limitations are that it only provides a 12-month forecast, and it only relies on 1 variable (the spread between the 10-year and 3-month Treasury rates).
- **Rabobank** has a recession probability model, but it is also only based on 1 variable (the spread between the 10-year and 1-year Treasury rates), and only covers one time period (17-months).
- **Wells Fargo Economics** has a few recession probability models that use a combination of economic and market data. However, they only limit their forecast to a 6-month horizon. [1]

Model Benchmarking / Comparison

Ideally, I would compare my model performance to each of the alternatives above. Currently, I cannot do this for the following reasons:

- **Guggenheim model**: Model performance data is not publicly released.
- **New York FED** model: Upon closer inspection, their model is built to answer the question "what is the probability that the U.S. will be in a recession X months from now?", whereas my model is built to answer the question "what is the probability that the U.S. will be in a recession within the next X months?".
- **Rabobank model**: The same reason. Additionally, the Rabobank model covers a 17-month time period, whereas my model covers 6-month time periods.
- **Moody's economists team** model: Model performance data is not publicly released.
- **Wells Fargo Economics** model: Model performance data is not publicly released. [1]
- **STATISTA** has a recession probability model from August 2020 to August 2025. Well paid access service.

2. Preparing the Data

2.1 Criteria according to which the data for the present study were collected.

Some things I had to consider when getting the data:

- **Economic data are released at different frequencies** (weekly, monthly, quarterly, etc.). To time-match data points, I settled for only using data that could be sampled monthly. As a result, all predictions must be conducted at a monthly frequency.
- **Varying data history lengths**. Some data has been released since 1919, while other data only goes back a few years. This means I had to exclude potentially useful data that just didn't have enough history.
- **Speaking of history**, I needed enough data to encompass as many recessions as I could. In the end, the full data set included 8 recessions since 1966.
- **Economic data gets revised often**. FRED (Federal Reserve Economic Data) does not provide the original figures. It only provides the revised figures (no matter how far after-the-fact those revisions are made).
- **Rare-event prediction**: Recessions are rare.

- **Small data set**: Because I am using economic data (which is updated at a frequency of months), I will only have a few hundred data points to work with.
- For practical reasons, I used mostly **public domain data** available through FRED and Stooq. I did not use potentially useful data that is stuck behind a paywall, such as The Conference Board Leading Economic Index.

2.2 Feature Selection.

Some project-specific considerations for feature selection:

- **Curse of Dimensionality**. Since the data set is so small (only a few hundred data points), one cannot include too many features in the final model. Otherwise, the model will fail to generalize to out-of-sample data. Therefore, features must be carefully chosen for the incremental value that each feature provides.
- **Domain knowledge is key**. Since the underlying process is a complex time series, automated feature selection methods have a high risk of over-fitting to prior data. Therefore, feature selection must be guided by a solid understanding of economic fundamentals.

A general outline for feature-selection process:

- Define the data set on which to perform exploratory data analysis (Jan 1962 to Dec 2012, Jan 2013 to Sep 2024) to ensure no intersection with cross-validation periods.
- Organize potential features into bucket, based on economic / theoretical characteristics (domain knowledge).
- We plot correlations between each individual feature and the target variable. - For final, we select features that have a low correlation with all that have already been "accepted" in the final dataset.

First, a sneak peek at the final feature list. Note that only 8 features made it to the final list:

- **Real-time Sahm Rule Recession Indicator**:

Sahm Recession Indicator signals the start of a recession when the three-month moving average of the national unemployment rate (U3) rises by 0.50 percentage points or more relative to the minimum of the three-month averages from the previous 12 months. This indicator is based on "real-time" data, that is, the unemployment rate (and the recent history of unemployment rates) that were available in a given month.

- **Industrial Production Index**:

The Industrial Production Index measures the output of the industrial sector, which is a key driver of economic growth. Changes in industrial production can signal shifts in overall economic activity and provide early warnings of potential recessions.

- **S&P 500 Index**:

The S&P 500 Index is a widely followed benchmark for the performance of the US stock market. Changes in the S&P 500 Index can reflect investor sentiment and expectations for future economic growth.

- **US 10-year Treasury Bond**:

The US 10-year Treasury bond yield is often seen as a safe-haven asset and can provide insights into investor expectations for future economic conditions. Changes in the yield curve can signal shifts in market sentiment and potential economic risks.

- **Slope of the yield curve**:

The slope of the yield curve, particularly the difference between short-term and long-term interest rates, can provide insights into market expectations for future economic conditions. Inverted yield curves, where short-term rates are higher than long-term rates, have historically been a reliable predictor of recessions.

- **Monthly Unemployment Rate**:

The monthly unemployment rate is a key indicator of labor market conditions and can provide insights into the overall health of the economy. Rising unemployment rates can signal economic weakness and potential recessions.

- **Personal consumption expenditure**:

Personal consumption expenditures are a key component of GDP and can provide insights into consumer spending patterns and overall economic activity. Changes in personal consumption expenditures can signal shifts in consumer confidence and potential risks for the economy.

- **Total Nonfarm Payroll**:

Total Nonfarm Payroll is a measure of the number of U.S. workers in the economy that excludes proprietors, private household employees, unpaid volunteers, farm employees, and the unincorporated self-employed.

This measure provides useful insights into the current economic situation because it can represent the number of jobs added or lost in an economy. Increases in employment might indicate that businesses are hiring which might also suggest that businesses are growing. Additionally, those who are newly employed have increased their personal incomes, which means their disposable incomes have also increased, thus fostering further economic expansion.

- **New Privately-Owned Housing Units Started: Total Units**:

New construction can reflect the state of the economy. A decrease in the number of construction starts can indicate that the economy is slowing or that there is low confidence in the real estate market, which can be an indicator of the onset of a recession.

An important feature:

In principle, **more features can be included** in the model, but **there is no time continuous data** for them since **January 1962**. Which would improve the model but reduce the range of data used to 1972 or later. There are also important **indicators**, which, however, are maintained as statistics with a **large time delay**. For this reason, they are not suitable for the present study. **Copyrighted data** are also not included in the study.

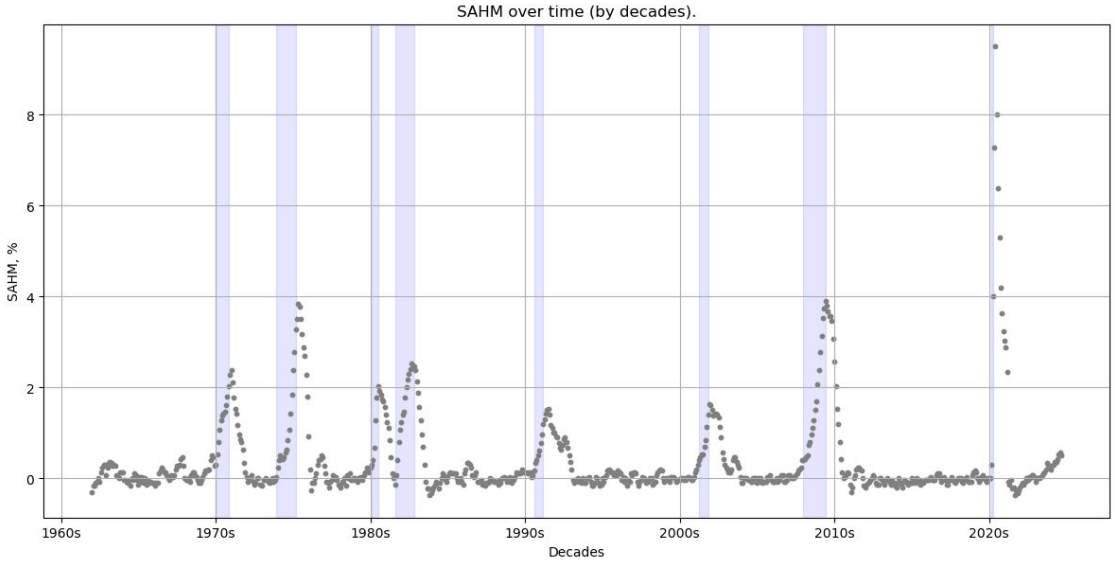
2.3 Data collection.

- **Real-time Sahm Rule Recession Indicator** (SAHMREALTIME):

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Real-time Sahm Rule Recession Indicator (SAHMREALTIME)**. [2] Units: Percentage Points, Seasonally Adjusted. Frequency: Monthly.

```
sahm = pd.read_csv('data/SAHM.csv') # Read the CSV file
sahm.columns = ["date", "sahm"]

def create_recession_chart():
    # Create the chart.
    plt.figure(figsize=(12, 6))
    plt.xlabel('Decades')
    plt.grid(True)
```

```
    plt.tight_layout()
    # Format the time axis to show only decades.
    decade_fmt = pd.to_datetime([f"{year}-01-01" for year in range(1960, 2030,
10)])
    plt.xticks(decade_fmt, labels=[f"{year}s" for year in range(1960, 2030, 10)
])

    # Add 10% blue for recession months.
    recession_periods = {
        (pd.Timestamp('1970-01-01'), pd.Timestamp('1970-11-01')),
        (pd.Timestamp('1973-12-01'), pd.Timestamp('1975-03-01')),
        (pd.Timestamp('1980-02-01'), pd.Timestamp('1980-07-01')),
        (pd.Timestamp('1981-08-01'), pd.Timestamp('1982-11-01')),
        (pd.Timestamp('1990-08-01'), pd.Timestamp('1991-03-01')),
        (pd.Timestamp('2001-04-01'), pd.Timestamp('2001-11-01')),
        (pd.Timestamp('2008-01-01'), pd.Timestamp('2009-06-01')),
        (pd.Timestamp('2020-03-01'), pd.Timestamp('2020-04-01'))
    }
    for start_date, end_date in recession_periods:
        plt.axvspan(start_date, end_date, color='blue', alpha=0.1)

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
sahm['date'] = pd.to_datetime(sahm['date'])
plt.scatter(sahm['date'], sahm['sahm'], s=10, color='gray')
# Readability settings.
plt.ylabel('SAHM, %')
plt.title('SAHM over time (by decades).')
plt.show()
```

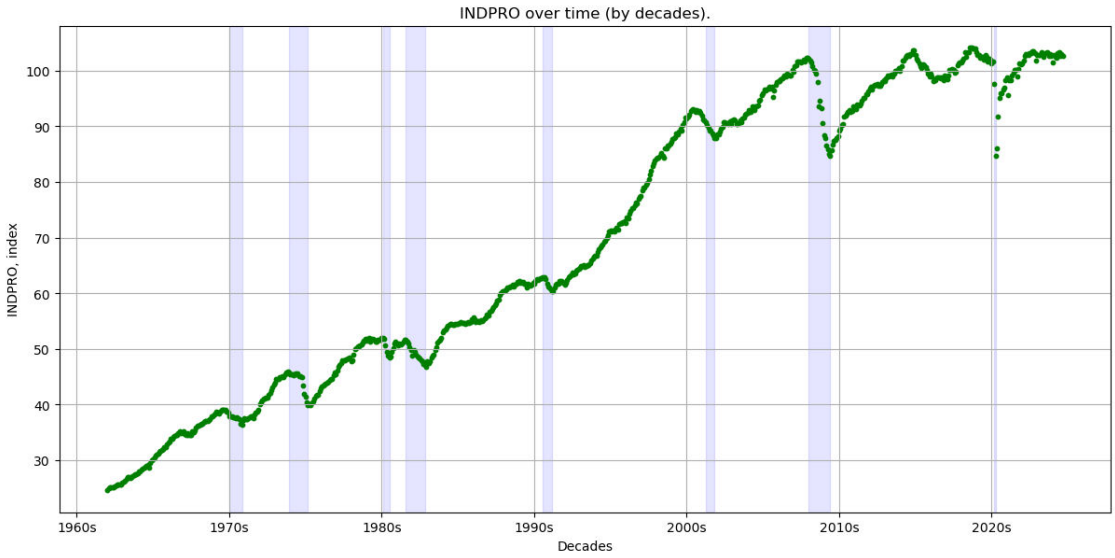


- **Industrial Production Index** (INDPRO)

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Industrial Production Index (INDPRO)** [3], Seasonally Adjusted. Frequency: Monthly.

```
indpro = pd.read_csv('data/INDPRO.csv') # Read the CSV file
indpro.columns = ["date", "indpro"]
```

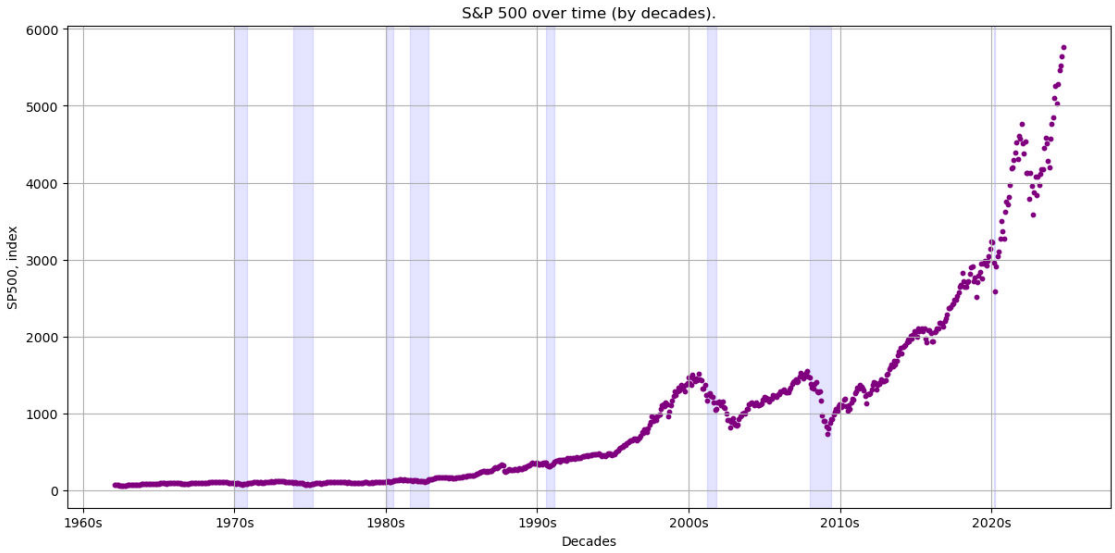
```
create_recession_chart()
# We make sure that the 'date' column is of type datetime.
indpro['date'] = pd.to_datetime(indpro['date'])
plt.scatter(indpro['date'], indpro['indpro'], s=10, color='green')
# Readability settings.
plt.ylabel('INDPRO, index')
plt.title('INDPRO over time (by decades).')
plt.show()
```



- **S&P 500 Index** (SP500)

From this site Stooq we download information for the period of our study: **S&P 500 Index (SP500)** [4], Frequency: Monthly. This is the only publicly available data for the S&P 500 Index, which contains information all the way back to 1962!

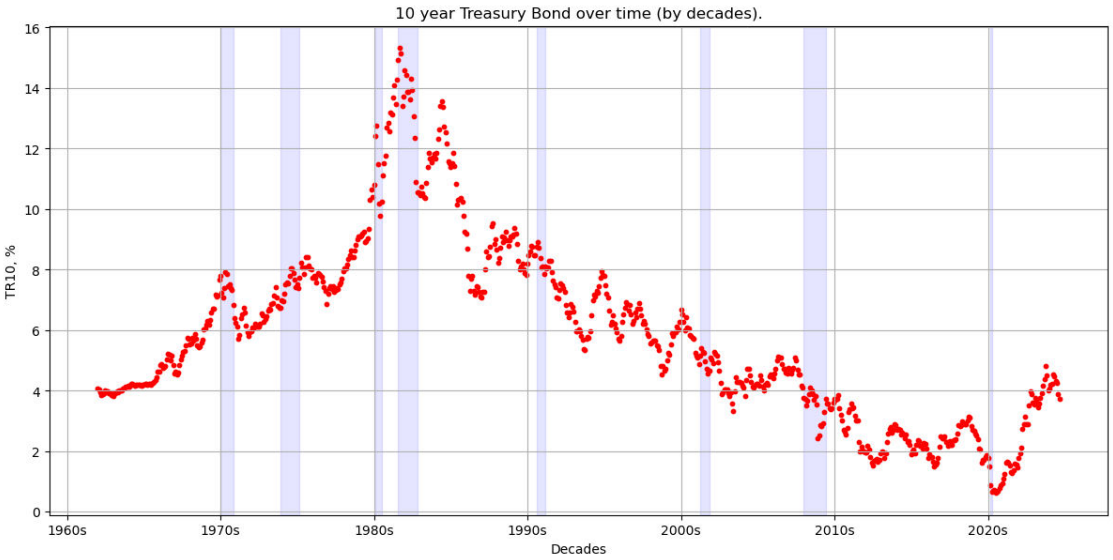
```
sp = pd.read_csv('data/SP500.csv') # Read the CSV file
# Keep only the required columns.
keep_col = ['Date', 'Close']
sp500 = sp[keep_col]
del sp
# Renaming columns.
sp500.columns = ['date', 'sp500']

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
sp500['date'] = pd.to_datetime(sp500['date'])
plt.scatter(sp500['date'], sp500['sp500'], s=10, color='purple')
# Readability settings.
plt.ylabel('SP500, index')
plt.title('S&P 500 over time (by decades).')
plt.show()
```



- **10 year Treasury Bond** (TR10)

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we download information for the period of our study: **10 year Treasury Bond** [5], Percent, Not Seasonally Adjusted. Frequency: **Daily**.

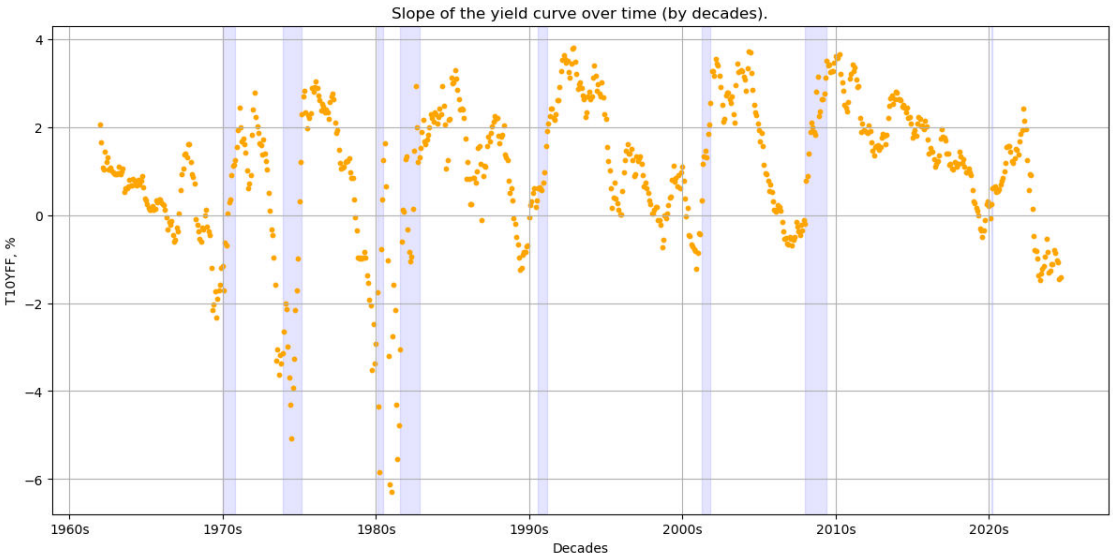
```
df = pd.read_csv('data/DGS10.csv') # Read the CSV file
df.columns = ["date", "tr10"]

# Replace '.' with NaN and convert column 'tr10' to numeric type.
df['tr10'] = pd.to_numeric(df['tr10'], errors='coerce')

# Group by month and calculate the arithmetic mean for each month.
df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace=True)
monthly_avg = df.resample('MS').mean()

# Create a new DF with the first number of the month and the arithmetic mean.
tr10 = monthly_avg.reset_index()
del df

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
tr10['date'] = pd.to_datetime(tr10['date'])
plt.scatter(tr10['date'], tr10['tr10'], s=10, color='red')
# Readability settings.
plt.xlabel('Decades')
plt.ylabel('TR10, %')
plt.title('10 year Treasury Bond over time (by decades).')
plt.show()
```



- **Slope of the yield curve** (T10YFF):

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we download information for the period of our study: **Slope of the yield curve (T10YFF)**. [6] Units: Percent, Seasonally Adjusted, Frequency: Monthly.

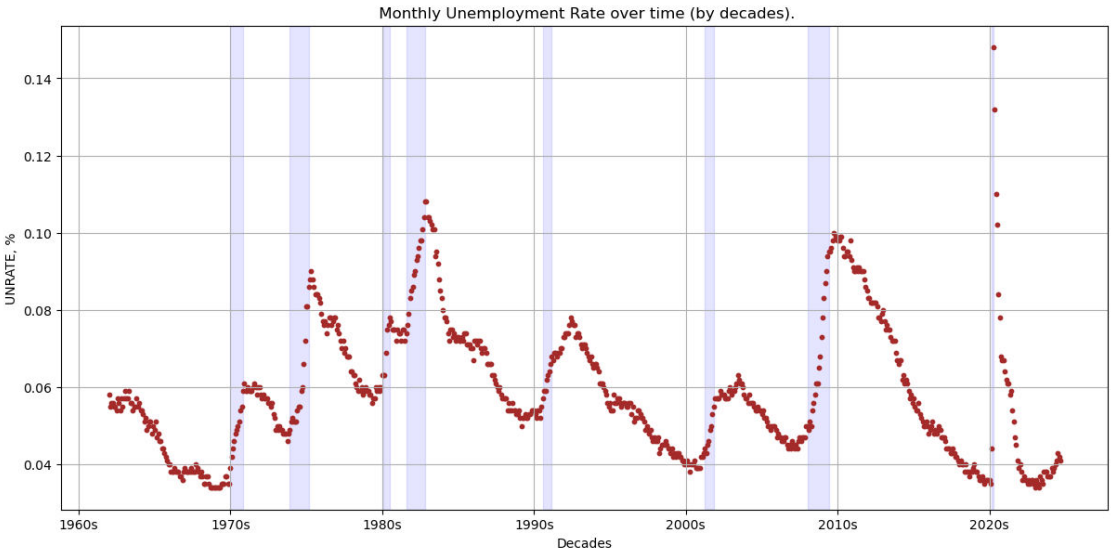
```
data = pd.read_csv('data/T10YFF.csv') # Read the CSV file
data.columns = ["date", "t10yff"]

# Replace '.' with NaN and convert column 't10yff' to numeric type.
data['t10yff'] = pd.to_numeric(data['t10yff'], errors='coerce')

# Group by month and calculate the arithmetic mean for each month.
data['date'] = pd.to_datetime(data['date'])
data.set_index('date', inplace=True)
monthly_avg = data.resample('MS').mean()

# Create a new DF with the first number of the month and the arithmetic mean.
t10yff = monthly_avg.reset_index()
del data

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
t10yff['date'] = pd.to_datetime(t10yff['date'])
plt.scatter(t10yff['date'], t10yff['t10yff'], s=10, color='orange')
# Readability settings.
plt.ylabel('T10YFF, %')
plt.title('Slope of the yield curve over time (by decades).')
plt.show()
```



- **Monthly Unemployment Rate** (UNRATE):

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we download information for the period of our study: **Monthly Unemployment Rate (UNRATE)**. [7] Units: Percent, Not Seasonally Adjusted. Frequency: Monthly.

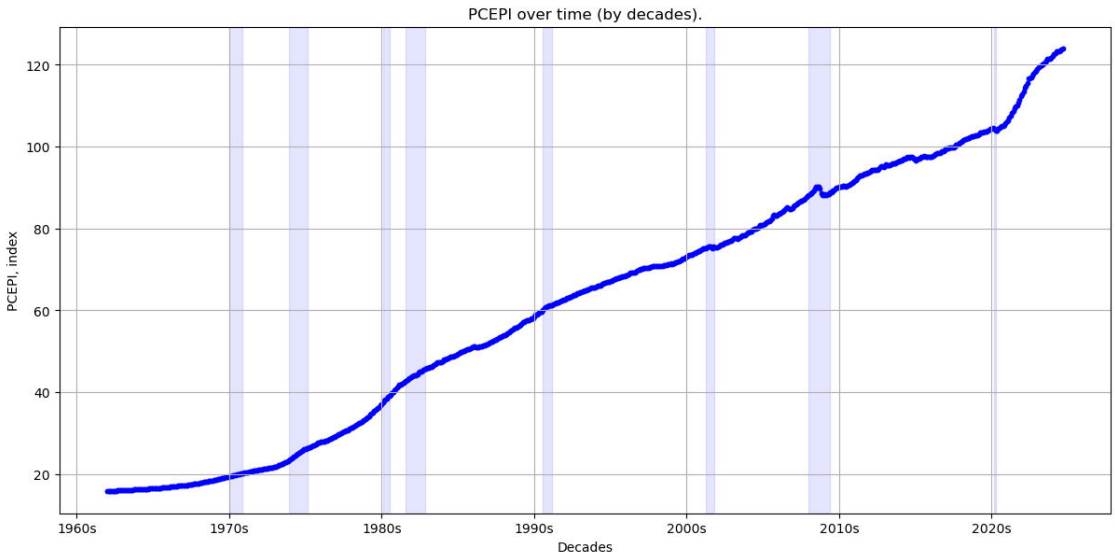
```
unrate = pd.read_csv('data/UNRATE.csv') # Read the CSV file
unrate.columns = ["date", "unrate"]
# => Conversion from % to numbers !!
unrate["unrate"] = unrate["unrate"] / 100

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
unrate['date'] = pd.to_datetime(unrate['date'])
plt.scatter(unrate['date'], unrate['unrate'], s=10, color='brown')
# Readability settings.
plt.ylabel('UNRATE, %')
plt.title(' Monthly Unemployment Rate over time (by decades).')
plt.show()
```



- **Personal Consumption Expenditures** (PCEPI)

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we download information for the period of our study: **Personal Consumption Expenditures (PCEPI)** [8], Seasonally Adjusted Annual Rate. Frequency: Monthly.

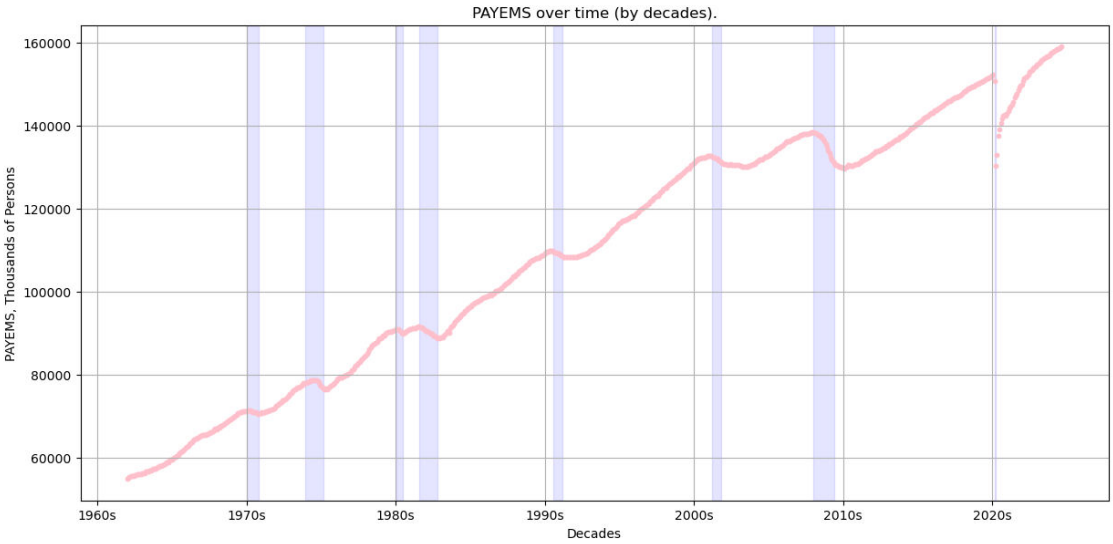
```
pcepi = pd.read_csv('data/PCEPI.csv') # Read the CSV file
pcepi.columns = ["date", "pcepi"]

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
pcepi['date'] = pd.to_datetime(pcepi['date'])
plt.scatter(pcepi['date'], pcepi['pcepi'], s=10, color='blue')
# Readability settings.
plt.ylabel('PCEPI, index')
plt.title('PCEPI over time (by decades).')
plt.show()
```



- **Total Nonfarm Payroll** (PAYEMS)

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we download information for the period of our study: **Total Nonfarm Payroll (PAYEMS)** [9], Units: Thousands of Persons, Seasonally Adjusted Annual Rate. Frequency: Monthly.

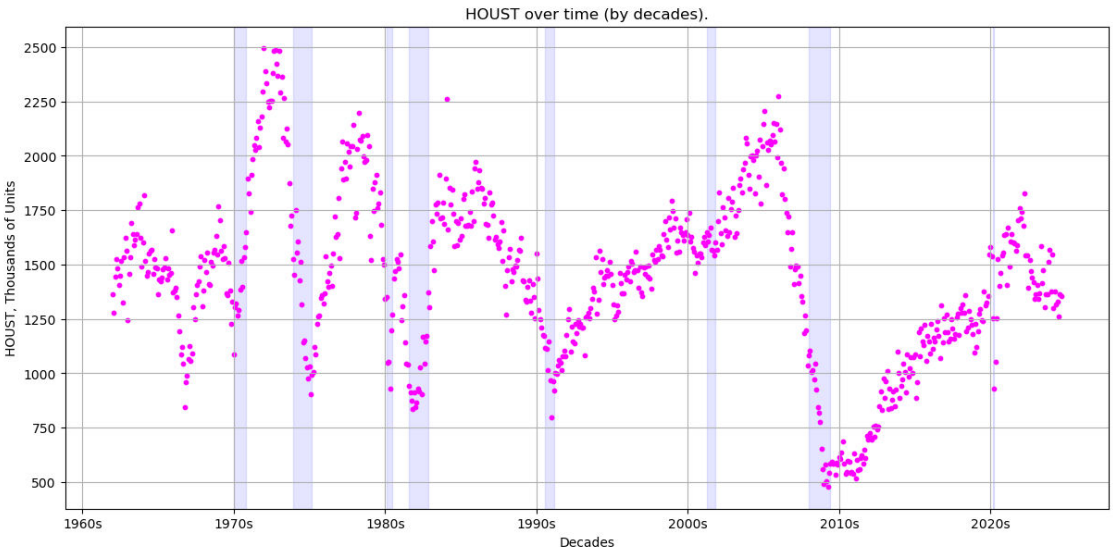
```
payems = pd.read_csv('data/PAYEMS.csv') # Read the CSV file
payems.columns = ["date", "payems"]

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
payems['date'] = pd.to_datetime(payems['date'])
plt.scatter(payems['date'], payems['payems'], s=10, color='pink')
# Readability settings.
plt.ylabel('PAYEMS, Thousands of Persons')
plt.title('PAYEMS over time (by decades).')
plt.show()
```



- **New Privately-Owned Housing Units Started: Total Units** (HOUST)

From this site [FEDERAL RESERVE BANK of ST. LOUIS] we download information for the period of our study: **New Privately-Owned Housing Units Started: Total Units (HOUST)** [9], Units: Thousands of Units, Seasonally Adjusted Annual Rate. Frequency: Monthly.

```
houst = pd.read_csv('data/HOUST.csv') # Read the CSV file
houst.columns = ["date", "houst"]

create_recession_chart()
# We make sure that the 'date' column is of type datetime.
houst['date'] = pd.to_datetime(houst['date'])
plt.scatter(houst['date'], houst['houst'], s=10, color='magenta')
# Readability settings.
plt.ylabel('HOUST, Thousands of Units')
plt.title('HOUST over time (by decades).')
plt.show()
```



- **Historical Recessions Data.**

With respect to the data collected above, I will **only consider data for recessions from 1962** to the present.

How are recessions defined?

The National Bureau of Economic Research (NBER) Business Cycle Dating Committee names “Peak” and “Trough” dates for business cycles. But there is a huge problem here:

`NBER declares “Peak” and “Trough” dates several months (sometimes years) after the fact!`

This means one: cannot label outputs one period at a time! So to deploy a recession predictor of any sort, one must re-train the model at each formal NBER “trough” announcement (after each recession end). Therefore, one must also take this same approach during backtesting! [1]

How do I label the class outputs?

At first, this seems easy: **Recession = 1**, **No Recession = 0**. That is true, but is insufficient as it pertains to the desired model scope. To determine the labeling process, one must figure out the question being answered.

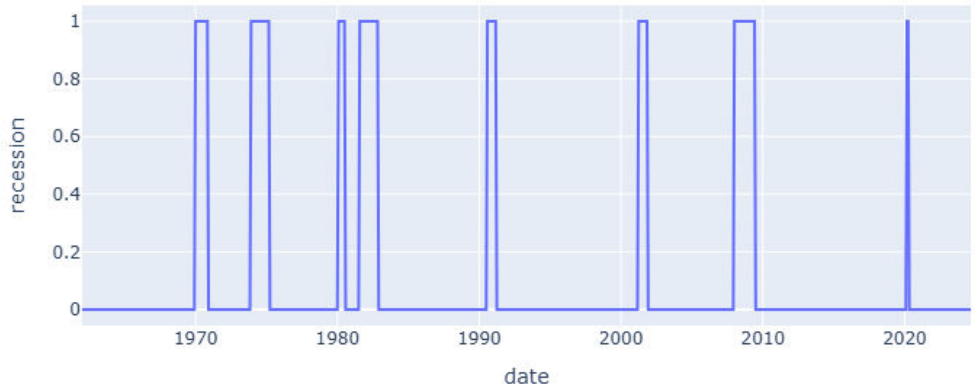
Is it: What is the probability that the U.S. will be in a recession X months from now?

If one wants to answer this question, one must forecast both the start and end of a recession. The model-predicted probabilities should start to drop before the recession actually ends. Needless to say, this is a pretty heroic task.

All months that are in a recessionary period will receive a label of “1”. All other months will receive a label of “0”.

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **NBER based Recession Indicators for the United States from the Period following the Peak through the Trough (USREC)**. [11] Unit: 1 or 0, Not Seasonally Adjusted. Frequency: Monthly.

```
recession = pd.read_csv('data/USREC.csv') # Read the CSV file
recession.columns = ["date", "recession"]

px.line(recession, x= recession.date, y='recession')
```




```
print("There are a total of {} months of recession in this data.".format(recession["recession"].sum()))
```

```
There are a total of 85 months of recession in this data.
```

Preparation of consolidated data:

```
# Returns an array where each value is the logarithmic difference between
# the current and previous values in the original array.
```

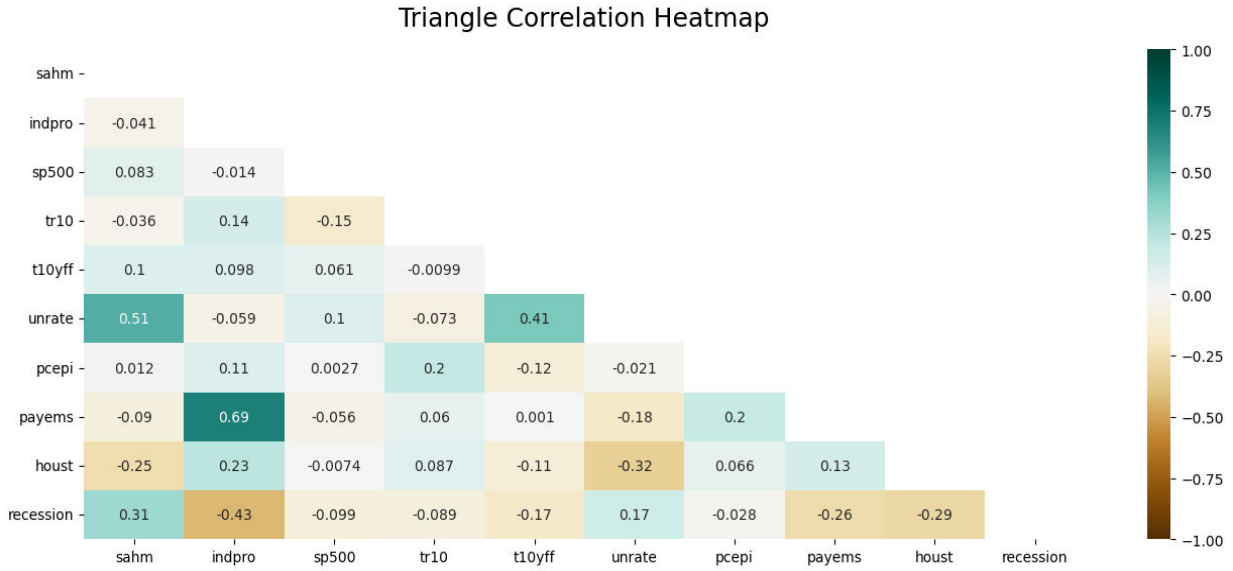
```
'''
    v.shift(1) => shifts the elements in the array down one position,
    with the first element becoming a NaN and the rest being moved forward one
position.
    v/v.shift(1) => calculates the ratio of the current value to the previous v
alue in the series.
    np.log(v/v.shift(1)) => calculates the natural logarithm (log to base e) of
this ratio.
'''
def log_diff(v):
    log_diff = np.log(v/v.shift(1))
    return log_diff

# Create a new empty table.
collected_data = pd.DataFrame(columns=[
    'sahm',     # Real-time Sahm Rule Recession Indicator
    'indpro',   # Industrial Production Index
    'sp500',    # S&P 500 Index
    'tr10',     # 10 year Treasury Bond
    't10yff',   # Slope of the yield curve
    'unrate',   # Unemployment Rate
    'pcepi',    # Personal Consumption Expenditures
    'payems',   # Total Nonfarm Payroll
    'houst',    #  New Privately-Owned Housing Units Started
    'recession' # Recession Indicators
])
# Add data to the columns and preprocessing.
collected_data['sahm'] = sahm['sahm']
collected_data['t10yff'] = t10yff['t10yff']
collected_data['unrate'] = unrate['unrate']
collected_data['recession'] = recession['recession']
# Logarithmization is appropriate for series that have exponential growth
# or high variation.
collected_data['indpro'] = log_diff(indpro['indpro'])
# Logarithmization => series that have exponential growth or high variation.
collected_data['sp500'] = log_diff(sp500['sp500'])
# TR10 has a clear trend (increasing or decreasing), differentiation can help.
collected_data['tr10'] = tr10['tr10'].diff()
# PCEPI has a clear trend (increasing or decreasing), differentiation can help.
collected_data['pcepi'] = pcepi['pcepi'].diff()
# PAYEMS has a clear trend (increasing or decreasing), differentiation can help
collected_data['payems'] = payems['payems'].diff()
# Time series normalization due to large residuals.
# We create an instance of MinMaxScaler.
scaler = MinMaxScaler()
# We apply MinMaxScaler to the 'payems' column.
collected_data['payems'] = scaler.fit_transform(collected_data[['payems']])
# We apply MinMaxScaler to the 'houst' column.
collected_data['houst'] = houst['houst']
collected_data['houst'] = scaler.fit_transform(collected_data[['houst']])

# Drop missing values after preprocessing process.
len_before = len(collected_data)
collected_data.dropna(inplace=True)
```

```
len_after = len(collected_data)
print("Dropped {} rows".format(len_before - len_after))
```

```
Dropped 1 rows
```

Since the Spearman and Kendall heatmaps are practically the same, we only look at one of them.

```
plt.figure(figsize=(16, 6)) # Determine the size of the figure.
# Define the mask to set the values in the upper triangle to True.
mask = np.triu(np.ones_like(collected_data.corr(method='kendall'), dtype=bool))
heatmap = sns.heatmap(collected_data.corr(), mask=mask, vmin=-1, vmax=1, annot=
True, cmap='BrBG')
heatmap.set_title('Triangle Correlation Heatmap', fontdict={'fontsize':18}, pad
=16)
plt.show()
```



Removing highly correlated features is a crucial preprocessing step in machine learning to improve model performance, interpretability, generalization, and computational efficiency. It helps create more robust and efficient models that are more suitable for real-world applications. [12]

```
# Create list of monthly dates.
dates = pd.date_range(start='02-1962', end='09-2024', freq='MS')
formatted_dates = pd.to_datetime(dates)

# Assuming collected_data is already an existing DataFrame (DF) with some data.
# Check the length of DF to ensure it matches the length of formatted_dates.
if len(collected_data) == len(formatted_dates):
    collected_data['date'] = formatted_dates
else:
    print("The length of collected_data does not match the length of formatted_
dates!")
collected_data.set_index('date', inplace=True)

collected_data # _______ DataFrame (DF) with all data! _______
```

```
            sahm    indpro     sp500      tr10    t10yff  unrate  pcepi  \
date
1962-02-01 -0.17  0.016229  0.016139 -0.043737  1.650556   0.055  0.042
1962-03-01 -0.17  0.005350 -0.005878 -0.108990  1.078182   0.056  0.021
```

```
1962-04-01 -0.10  0.002130 -0.063973 -0.087455  1.043000   0.056  0.018
1962-05-01 -0.07 -0.001066 -0.089914  0.030636  1.441818   0.055  0.010
1962-06-01  0.00 -0.002132 -0.085381  0.035411  1.206667   0.055  0.010
...          ...       ...       ...       ...       ...     ...    ...
2024-05-01  0.37  0.006954  0.046904 -0.056818 -0.847727   0.040 -0.010
2024-06-01  0.43  0.001500  0.034082 -0.177010 -1.024737   0.041  0.145
2024-07-01  0.53 -0.006214  0.011258 -0.056627 -1.081364   0.043  0.202
2024-08-01  0.57  0.003373  0.022578 -0.377727 -1.459091   0.042  0.142
2024-09-01  0.50 -0.002832  0.019996 -0.147409 -1.406500   0.041  0.217

              payems     houst  recession
date
1962-02-01  0.827913  0.396825          0
1962-03-01  0.819544  0.478671          0
1962-04-01  0.829109  0.518849          0
1962-05-01  0.817113  0.498512          0
1962-06-01  0.816714  0.459325          0
...              ...       ...        ...
2024-05-01  0.824685  0.415179          0
2024-06-01  0.820780  0.422123          0
2024-07-01  0.821816  0.388889          0
2024-08-01  0.822414  0.437996          0
2024-09-01  0.826200  0.434524          0

[752 rows x 10 columns]
```

```
collected_data.to_csv(r'data/collected_data.csv') # We save the DF to a file.
```

To find out whether a time series is stationary or not, we use the Dickey-Fuller test. From the **calculations.ipynb** file (See chapter: Appendices), we collect data about the behavior of the time series in the following table:

Augmented Dickey-Fuller tests and conclusions

Base indicator	Index	Conclusion
Real-time Sahm Rule Recession Indicator	SAHM	stationary
Industrial Production Index	INDPRO	stationary
S&P 500 Index	SP500	stationary
10 year Treasury Bond	TR10	stationary
Slope of the yield curve	T10YFF	stationary
Monthly Unemployment Rate	UNRATE	stationary
Personal Consumption Expenditures	PCEPI	stationary
Total Nonfarm Payroll	PAYEMS	stationary
New Privately-Owned Housing Units Started	HOUST	stationary

```
collected_data.describe().T
```

```
           count      mean       std       min       25%       50%       75%  \
sahm       752.0  0.412527  0.984315 -0.370000 -0.030000  0.030000  0.370000
indpro     752.0  0.001897  0.009524 -0.142045 -0.001726  0.002446  0.006304
sp500      752.0  0.005887  0.043518 -0.245428 -0.017919  0.009800  0.034384
tr10       752.0 -0.000478  0.277827 -1.755317 -0.145981 -0.002482  0.149554
t10yff     752.0  0.982359  1.619839 -6.291905  0.118571  1.157828  2.181648
unrate     752.0  0.058871  0.017200  0.034000  0.046000  0.056000  0.070000
pcepi      752.0  0.143822  0.156321 -1.058000  0.057750  0.131000  0.209000
payems     752.0  0.821600  0.032212  0.000000  0.818827  0.823410  0.826967
houst      752.0  0.475000  0.192394  0.000000  0.358879  0.489087  0.584325
recession  752.0  0.113032  0.316843  0.000000  0.000000  0.000000  0.000000

                max
sahm       9.500000
indpro     0.063771
sp500      0.151043
tr10       1.612464
t10yff     3.803636
unrate     0.148000
pcepi      1.089000
payems     1.000000
houst      1.000000
recession  1.000000
```

Looks good!

3. Modeling

3.1 Class Imbalance

First, we might ask ourselves, how frequently do these recession months occur in the data?

```
c1 = collected_data["recession"].value_counts()[0]
c2 = collected_data["recession"].value_counts()[1]
print(collected_data["recession"].value_counts())
print("The minority class only makes up {} % ".format(round(c2/c1 * 100, 2)))
```

```
recession
0    667
1     85
Name: count, dtype: int64
The minority class only makes up 12.74 %
```

Type of periods	Start	Stop	Recessions
Train	02.1962	12.2012	Yes, 7
Test	01.2013	09.2024	Yes, 1

So how can you handle class imbalances? We will **only use those classification models that deal well with unbalanced classes**.

Without a validation set, we will not be able to evaluate the performance of the model during training and tune the hyperparameters. Although the data is not much, creating new "synthetic" data (oversample using SMOTE) will not lead to precision in the final result. In view of the fact that we have few training records, **k-fold cross-validation** is the appropriate method. Usually k=5 is a good choice.

When dealing with imbalanced data, especially in a time series context, it is important to approach model estimation carefully. We evaluate the models on the validation set to get an idea of their performance. We use metrics that are appropriate for unbalanced data, such as **F1-score**, **AUC**, **Precision**, and **Recall**, instead of just **Accuracy**. Even in the context of unbalanced data, precision is a more appropriate metric for evaluating model performance than accuracy.

3.2 Modeling

The evaluation criteria, the **models and their settings** are described in the files: modeling_1.ipynb and modeling_2.ipynb. (See chapter: Appendices) Here we will only visualize the final result of the validation:

Best models are: Random Forest, XGBoost and CatBoost

Model	F1-score	Precision	Recall	Accuracy
Logistic Regression	0.8024	0.7976	0.8072	0.9460
Decision Tree	0.8075	0.8333	0.7831	0.9493
Random Forest	0.8842	0.8926	0.8787	0.9689
XGBoost	0.8557	0.9267	0.8088	0.9657
CatBoost	0.8675	0.8575	0.9051	0.9624
SVM	0.7677	0.7810	0.7978	0.9330

Analysis based on model validation:

Based on overall performance, **Random Forest** is the best model, followed by **CatBoost** and **XGBoost**. **Decision Tree** and **Logistic Regression** also show good performance, but **SVM** is the weakest model in all metrics. That's why it drops out when testing the models.

3.3 Testing the models

5 ML **models** should be **trained** and **tested**. AUC, F1-score, Precision, Recall, common confusion_matrix and Specificity will be calculated for each model. A ROC Curve plot will be drawn for the four models.

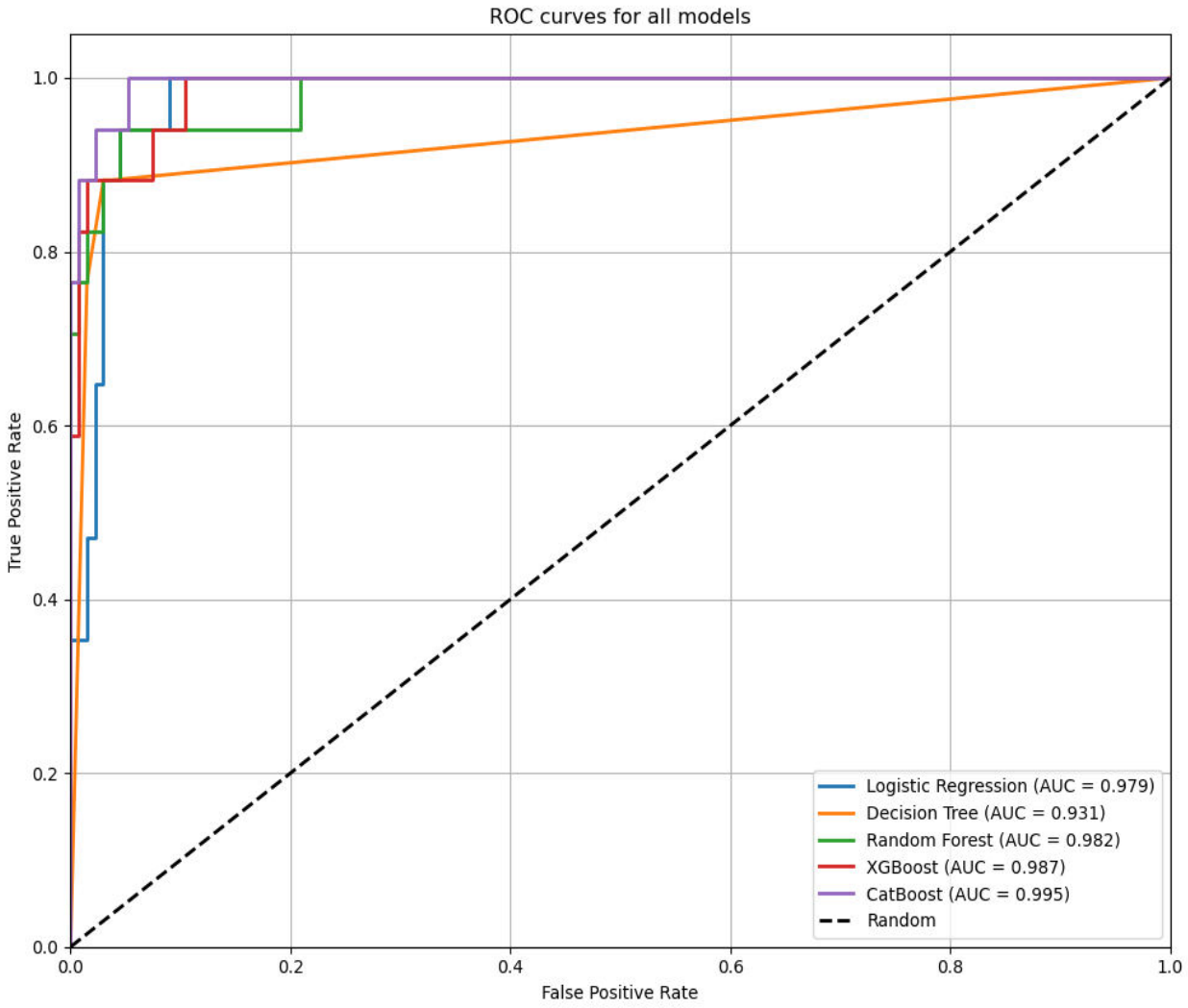
```
# get X and y
X = collected_data.drop(['recession'], axis=1)
y = collected_data['recession']

# We define the training period.
X_train, y_train = X.loc["1962-02-01":"2012-12-01"], y.loc["1962-02-01":"2012-12-01"]
# We define the test period.
X_test, y_test = X.loc["2013-01-01":], y.loc["2013-01-01":]

# Proper separation of data with stratification.
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y  # This is the key change.
)

print("Data form:")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
```

```
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

print("\nDistribution of classes:")
print("Train class distribution:")
print(pd.Series(y_train).value_counts(normalize=True))
print("\nTest class distribution:")
print(pd.Series(y_test).value_counts(normalize=True))

# Scaling the data.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Setting the graphics style.
plt.rcParams['figure.figsize'] = (10, 8)
plt.rcParams['axes.grid'] = True

# A function to calculate specificity.
def specificity_score(y_true, y_pred):
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    return tn / (tn + fp)

# Initialization of models with their hyperparameters.
models = {
    'Logistic Regression': {
        'model': LogisticRegression(
            C=1,
            class_weight={0: 1, 1: 6},
            max_iter=1000,
            penalty='l2',
            solver='saga',
            random_state=42
        ),
        'needs_scaling': True
    },
    'Decision Tree': {
        'model': DecisionTreeClassifier(
            criterion='entropy',
            max_depth=5,
            max_features=None,
            min_samples_leaf=1,
            min_samples_split=2,
            random_state=42
        ),
        'needs_scaling': False
    },
    'Random Forest': {
        'model': RandomForestClassifier(
            bootstrap=False,
            class_weight='balanced',
            max_depth=15,
            max_features='sqrt',
```

```
            min_samples_leaf=4,
            min_samples_split=10,
            n_estimators=300,
            random_state=42
        ),
        'needs_scaling': False
    },
    'XGBoost': {
        'model': xgb.XGBClassifier(
            colsample_bytree=0.9,
            gamma=0.1,
            learning_rate=0.05,
            max_depth=6,
            min_child_weight=3,
            n_estimators=100,
            scale_pos_weight=1,
            subsample=1.0,
            random_state=42
        ),
        'needs_scaling': False
    },
    'CatBoost': {
        'model': CatBoostClassifier(
            auto_class_weights='Balanced',
            bagging_temperature=0,
            border_count=32,
            depth=8,
            iterations=200,
            l2_leaf_reg=1,
            learning_rate=0.05,
            random_strength=10,
            random_state=42,
            verbose=False
        ),
        'needs_scaling': False
    }
}

# List to store the results.
results = []
# Creating a figure for the ROC curves.
fig, ax = plt.subplots()

# Training and evaluation of models.
for name, model_info in models.items():
    model = model_info['model']
    needs_scaling = model_info['needs_scaling']

    # Choosing the right data according to the scaling need.
    X_train_data = X_train_scaled if needs_scaling else X_train
    X_test_data = X_test_scaled if needs_scaling else X_test

    try:
```

```
        # Training the model.
        model.fit(X_train_data, y_train)

        # Predictions:
        y_pred = model.predict(X_test_data)
        y_pred_proba = model.predict_proba(X_test_data)[:, 1]

        # Checking predictions.
        print(f"\nМодел: {name}")
        print("Unique values in predictions:", np.unique(y_pred))
        print("Range of probabilities:", np.min(y_pred_proba), "-", np.max(y_pr
ed_proba))

        # Confusion matrix:
        tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
        print(f"Confusion Matrix: TN={tn}, FP={fp}, FN={fn}, TP={tp}")

        # Calculating metrics:
        fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
        roc_auc = auc(fpr, tpr)

        # Save metrics.
        results.append({
            'Model': name,
            'AUC': roc_auc,
            'F1-score': f1_score(y_test, y_pred),
            'Precision': precision_score(y_test, y_pred),
            'Recall': recall_score(y_test, y_pred),
            'Specificity': specificity_score(y_test, y_pred)
        })

        # Add ROC curve.
        ax.plot(fpr, tpr, lw=2, label=f'{name} (AUC = {roc_auc:.3f})')

    except Exception as e:
        print(f"Error at {name}: {str(e)}")

# Finalizing the ROC plot.
ax.plot([0, 1], [0, 1], 'k--', label='Random', lw=2)
ax.set_xlim([0.0, 1.0])
ax.set_ylim([0.0, 1.05])
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('True Positive Rate')
ax.set_title('ROC curves for all models')
ax.legend(loc="lower right")
ax.grid(True)
plt.tight_layout()
plt.show()

# Create a DataFrame with the results.
results_df = pd.DataFrame(results)
results_df = results_df.set_index('Model')
print("\nMetrics for all models:")
print(results_df.round(4).to_string())
```

```
Data form:
X_train shape: (601, 9)
X_test shape: (151, 9)
y_train shape: (601,)
y_test shape: (151,)

Distribution of classes:
Train class distribution:
recession
0    0.886855
1    0.113145
Name: proportion, dtype: float64

Test class distribution:
recession
0    0.887417
1    0.112583
Name: proportion, dtype: float64

Model: Logistic Regression
Unique values in predictions: [0 1]
Range of probabilities: 2.5377559223161707e-07 - 0.999983908938828
Confusion Matrix: TN=122, FP=12, FN=0, TP=17

Model: Decision Tree
Unique values in predictions: [0 1]
Range of probabilities: 0.0 - 1.0
Confusion Matrix: TN=130, FP=4, FN=2, TP=15

Model: Random Forest
Unique values in predictions: [0 1]
Range of probabilities: 0.0 - 0.9962633130021159
Confusion Matrix: TN=130, FP=4, FN=3, TP=14

Model: XGBoost
Unique values in predictions: [0 1]
Range of probabilities: 0.002605706 - 0.9757932
Confusion Matrix: TN=132, FP=2, FN=2, TP=15

Model: CatBoost
Unique values in predictions: [0 1]
Range of probabilities: 7.03488963512961e-05 - 0.9993221405581604
Confusion Matrix: TN=130, FP=4, FN=1, TP=16
```



Metrics for all models:

Model	AUC	F1-score	Precision	Recall	Specificity
Logistic Regression	0.9794	0.7391	0.5862	1.0000	0.9104
Decision Tree	0.9306	0.8333	0.7895	0.8824	0.9701
Random Forest	0.9820	0.8000	0.7778	0.8235	0.9701
XGBoost	0.9868	0.8824	0.8824	0.8824	0.9851
CatBoost	0.9947	0.8649	0.8000	0.9412	0.9701

F1-scores are balanced. Most models reach 0.8824. Decision Tree is slightly weaker at 0.8333. Logistic Regression is weakest at 0.7727.

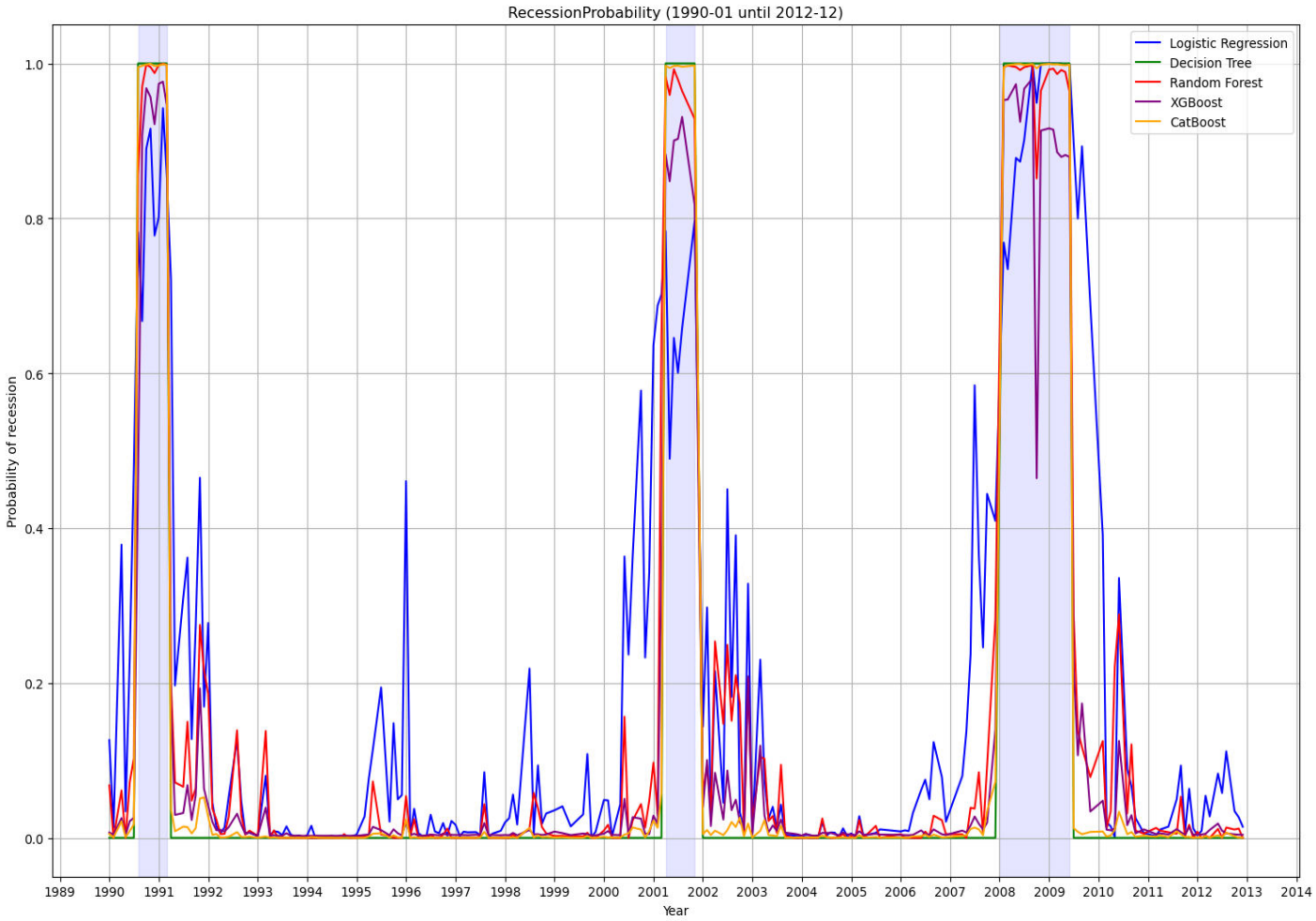
Precision and **Recall** are balanced for most models (0.8824), which is a good sign. **Specificity** was high for all models (>0.92), indicating good recognition of negative cases.

We have to choose a best model:

CatBoost seems to be the best choice with the highest AUC. **XGBoost** and **Random Forest** are very close to it in terms of performance. All **three have the same metrics** for F1-score, Precision, Recall and Specificity. Finding the best model or models in this case can hardly be done just by the numbers. Let's see how the models behave in "real time" around the time of economic recession. The behavior of the models is graphically depicted. We also have 3 recessionary periods in the form of light blue columns.

```
# We create a DataFrame with predictions using the X_train index.
predictions_df = pd.DataFrame(index=X_train.index)
# We add the predictions from each model.
for name, model_info in models.items():
    model = model_info['model']
    needs_scaling = model_info['needs_scaling']

    X_data = X_train_scaled if needs_scaling else X_train
    predictions = model.predict_proba(X_data)[:, 1]

    predictions_df[name] = predictions

# We sort the index to ensure it is monotonic.
predictions_df = predictions_df.sort_index()

# We find the closest date to the beginning and end of the desired period.
start_date = predictions_df.index[predictions_df.index >= '1990-01-01'].min()
end_date = predictions_df.index[predictions_df.index <= '2012-12-31'].max()

# We filter the data only for the period 1990-2012.
predictions_df = predictions_df.loc[start_date:end_date]

# We create the graphic.
plt.figure(figsize=(15, 10))
# Colors for every model.
colors = ['blue', 'green', 'red', 'purple', 'orange']

# We visualize the predictions for each model.
for (name, color) in zip(models.keys(), colors):
    plt.plot(predictions_df.index, predictions_df[name], color=color, label=nam
e)

# We add vertical columns for the specific periods.
periods = [
    ('1990-08-01', '1991-03-01'),
    ('2001-04-01', '2001-11-01'),
    ('2008-01-01', '2009-06-01')
]

for start, end in periods:
    plt.axvspan(pd.to_datetime(start), pd.to_datetime(end), color='blue', alpha
=0.1)

plt.title(f'RecessionProbability ({start_date.strftime("%Y-%m")} until {end_dat
e.strftime("%Y-%m")})')
plt.xlabel('Year')
plt.ylabel('Probability of recession')
plt.legend()
plt.grid(True)

# Format x-axis to show only years.
plt.gca().xaxis.set_major_locator(YearLocator())
plt.gca().xaxis.set_major_formatter(DateFormatter('%Y'))
plt.tight_layout()
```

```
plt.show()
```



The "cleanest" is the Decision Tree model, but here we only have 3 recession periods depicted. It is followed by the **CatBoost** model. The latter performs **best as numerical metrics**, so it is **chosen as the model that will predict** future economic recession.

4. Final forecast

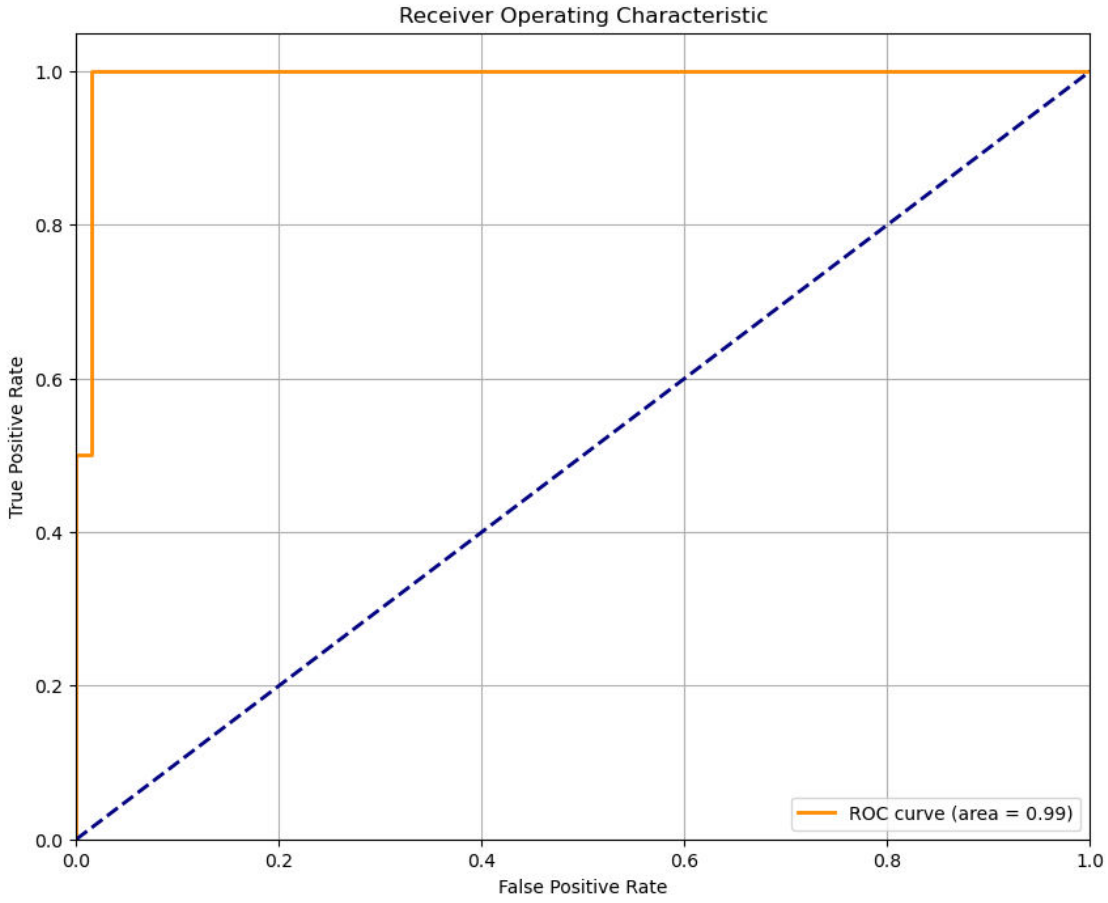
4.1 First way to forecast.

Let's start calculating the probability of the US going into recession now.

```
collected_data = pd.read_csv('data/collected_data.csv') # Read the CSV file
data = collected_data.copy()
# Setting column 'date' as index.
data = data.set_index('date', drop=True)
# get X and y
X = data.drop(['recession'], axis=1)
y = data['recession']

# We define the training period.
X_train, y_train = X.loc["1962-02-01":"2012-12-01"], y.loc["1962-02-01":"2012-12-01"]
# We define the test period.
X_test, y_test = X.loc["2013-01-01":], y.loc["2013-01-01":]

# Building and training the model.
```

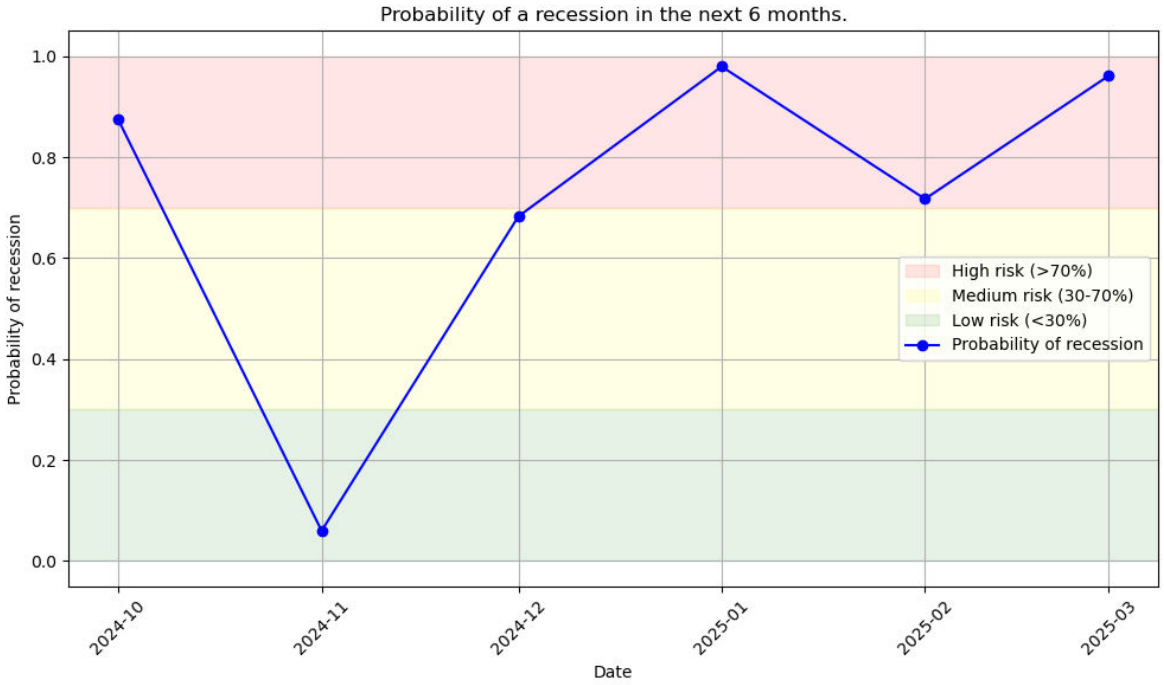
```
cb_model = CatBoostClassifier(
    auto_class_weights='Balanced',
    iterations=200,
    depth=8,
    learning_rate=0.05,
    random_state=42,
    verbose=False
)

# Training the model.
cb_model.fit(X_train, y_train)

# Using the prediction model.
predictions = cb_model.predict(X_test)

# Recession probabilities (ie, how likely a recession is 6 months from now).
# Probability of Class "1" (Recession).
probabilities = cb_model.predict_proba(X_test)[:, 1]

# Model evaluation:
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)
roc_auc = roc_auc_score(y_test, probabilities)

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
print(f"ROC AUC: {roc_auc:.4f}")

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, predictions))

print("\nClassification Report:")
print(classification_report(y_test, predictions))

# ROC curve visualization.
fpr, tpr, thresholds = roc_curve(y_test, probabilities)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc
:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```

```
Accuracy: 0.9645
Precision: 0.2857
Recall: 1.0000
F1 Score: 0.4444
ROC AUC: 0.9928

Confusion Matrix:
[[134   5]
 [  0   2]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.96      0.98       139
           1       0.29      1.00      0.44         2

    accuracy                           0.96       141
   macro avg       0.64      0.98      0.71       141
weighted avg       0.99      0.96      0.97       141
```




```
# We use the latest available data (for example, the last 6 months).
# Will be our current "input" for the model to make a forecast 6 months ahead.
latest_data = X.tail(6)  # We take the last 6 months of available data.

# We predict the probability of a recession 6 months ahead.
# Probability of class "1" (recession).
future_probabilities = cb_model.predict_proba(latest_data)[:, 1]
```

```
# We create the correct future dates starting from October 2024.
start_date = pd.Timestamp('2024-10-01')

# MS means start of month.
future_dates = pd.date_range(start=start_date, periods=6, freq='MS')

# Visualization of the likelihood of a recession in the next 6 months.
plt.figure(figsize=(10, 6))

# We are adding color zones for the different risk levels.
plt.axhspan(0.7, 1.0, color='red', alpha=0.1, label='High risk (>70%)')
plt.axhspan(0.3, 0.7, color='yellow', alpha=0.1, label='Medium risk (30-70%)')
plt.axhspan(0, 0.3, color='green', alpha=0.1, label='Low risk (<30%)')

plt.plot(future_dates, future_probabilities, marker='o', linestyle='-', color='
b', label='Probability of recession')

# Format x-axis dates.
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))

plt.xlabel('Date')
plt.ylabel('Probability of recession')
plt.title('Probability of a recession in the next 6 months.')
plt.xticks(future_dates, rotation=45)
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

# Probability of a recession in the next months.
future_prob_df = pd.DataFrame({
    'date': future_dates,
    'probability_of_recession': future_probabilities
})
future_prob_df.set_index('date', inplace=True)

future_prob_df
```




```
            probability_of_recession
date
2024-10-01                  0.873895
2024-11-01                  0.059752
2024-12-01                  0.682443
2025-01-01                  0.979363
2025-02-01                  0.717093
2025-03-01                  0.961044
```

```
# Probab. of recession => as arithmetic mean of probabilities for all months.
average_probability = (future_prob_df['probability_of_recession'].mean() * 100)
.round(2)
formatted_probability = f"{average_probability} %"
print(">>> ------ >> ----- > ----")
print(f"Average probability of a recession over the entire period: {formatted_p
robability}")
```

```
>>> ------ >> ----- > ----
Average probability of a recession over the entire period: 71.23 %
```

Have we solved the task yet? Not quite. Let's check with another way to predict.

*4.2 **Second way** of prediction.*

Here we use a method to predict each feature using **ARIMA**. Thus, after for each 1 economic indicator we predict its behavior for another 6 months, we collect everything again in a whole dataframe and run the model again. It will have the added information for its own and we can easily visualize the value of the target variable: recession.

In time series analysis used in statistics and econometrics, **autoregressive integrated moving average (ARIMA)** and seasonal ARIMA (SARIMA) models are generalizations of the autoregressive moving average (ARMA) model to non-stationary series and periodic variation, respectively. All these models are fitted to time series in order to better understand it and predict future values. The purpose of these generalizations is to fit the data as well as possible. Specifically, ARMA assumes that the series is stationary, that is, its expected value is constant in time. If instead the series has a trend (but a constant

variance/autocovariance), the trend is removed by "differencing", leaving a stationary series. This operation generalizes ARMA and corresponds to the "integrated" part of ARIMA. [13]

The code for the ARIMA settings and the process of generating the new data can be seen in this file (See chapter: Appendices): ARIMA.ipynb

We extract the new dataframe with the new data added, ARIMA prediction:

```
# We read the CSV file with the newly generated data.
new_data = pd.read_csv('data/combined_data.csv')
# Setting column 'Unnamed: 0' as index.
new_data = new_data.rename(columns={'Unnamed: 0': 'date'})
# We make the 'date' column an index.
new_data = new_data.set_index('date')

new_data
```

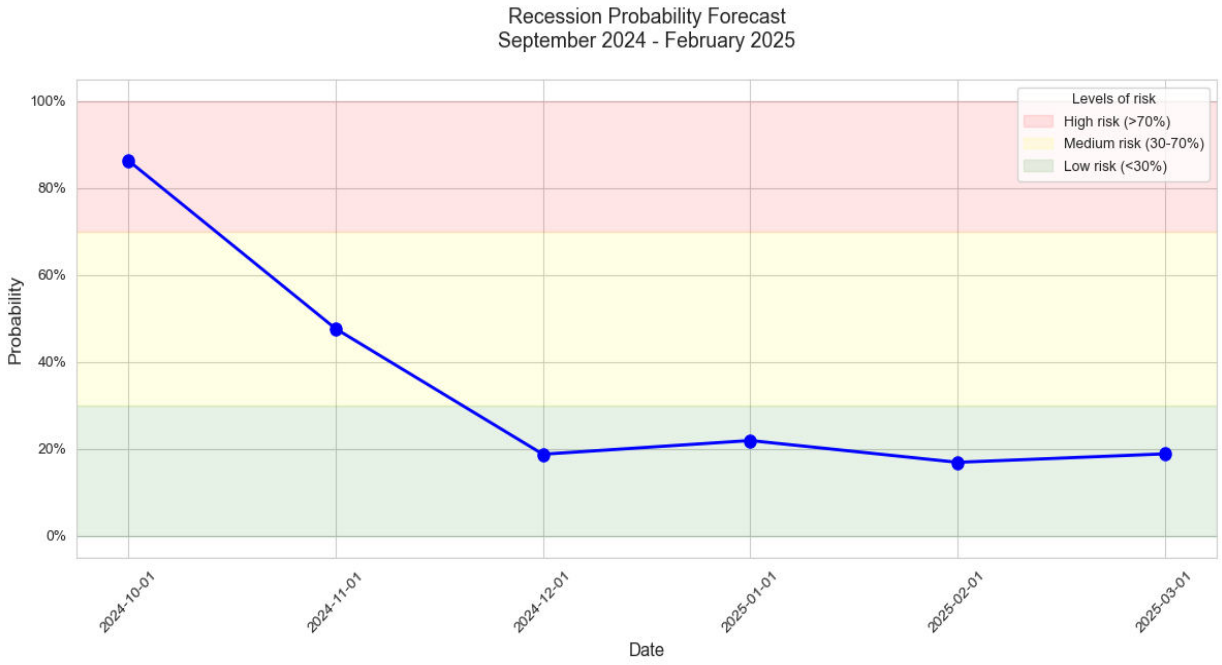
```
                sahm    indpro     sp500      tr10    t10yff    unrate  \
date
1962-02-01 -0.170000  0.016229  0.016139 -0.043737  1.650556  0.055000
1962-03-01 -0.170000  0.005350 -0.005878 -0.108990  1.078182  0.056000
1962-04-01 -0.100000  0.002130 -0.063973 -0.087455  1.043000  0.056000
1962-05-01 -0.070000 -0.001066 -0.089914  0.030636  1.441818  0.055000
1962-06-01  0.000000 -0.002132 -0.085381  0.035411  1.206667  0.055000
...              ...       ...       ...       ...       ...       ...
2024-11-01  0.371263  0.001896 -0.000628  0.000468 -1.054334  0.042184
2024-12-01  0.365931  0.001896  0.012222 -0.000439 -0.907669  0.042725
2025-01-01  0.369188  0.001896 -0.000270 -0.000439 -0.771676  0.043291
2025-02-01  0.373684  0.001896  0.011875 -0.000439 -0.645579  0.043803
2025-03-01  0.377939  0.001896  0.000068 -0.000439 -0.528658  0.044322

               pcepi    payems     houst  recession
date
1962-02-01  0.042000  0.827913  0.396825          0
1962-03-01  0.021000  0.819544  0.478671          0
1962-04-01  0.018000  0.829109  0.518849          0
1962-05-01  0.010000  0.817113  0.498512          0
1962-06-01  0.010000  0.816714  0.459325          0
...              ...       ...       ...        ...
2024-11-01  0.199241  0.820944  0.434200          0
2024-12-01  0.196893  0.821594  0.430898          0
2025-01-01  0.194642  0.821594  0.435397          0
2025-02-01  0.192485  0.821594  0.432419          0
2025-03-01  0.190417  0.821594  0.436556          0

[758 rows x 10 columns]
```

We train the model on the new dataframe and test it on the new test set containing the generated new 6 months.

```
# get X and y
X = new_data.drop(['recession'], axis=1)
y = new_data['recession']

# We define the training period.
```

```
X_train, y_train = X.loc["1962-02-01":"2012-12-01"], y.loc["1962-02-01":"2012-12-01"]
# We define the test period.
X_test, y_test = X.loc["2013-01-01":], y.loc["2013-01-01":]

# Building and training the model.
new_cb_model = CatBoostClassifier(
    auto_class_weights='Balanced',
    iterations=200,
    depth=8,
    learning_rate=0.05,
    random_state=42,
    verbose=False
)

# Training the model.
new_cb_model.fit(X_train, y_train)

# We make predictions for the entire test period.
all_probabilities = new_cb_model.predict_proba(X_test)[:, 1]

# We create a DataFrame with all predictions.
full_results_df = pd.DataFrame({
    'Date': X_test.index,
    'Recession Probability': all_probabilities
})

# We only take the last 6 months for preview.
results_df = full_results_df.tail(6)

# We create the visualization.
plt.figure(figsize=(12, 6))
sns.set_style("whitegrid")
plt.plot(results_df['Date'], results_df['Recession Probability'],
         marker='o', linewidth=2, markersize=8, color='blue')

# We are adding color zones for the different risk levels.
plt.axhspan(0.7, 1.0, color='red', alpha=0.1, label='High risk (>70%)')
plt.axhspan(0.3, 0.7, color='yellow', alpha=0.1, label='Medium risk (30-70%)')
plt.axhspan(0, 0.3, color='green', alpha=0.1, label='Low risk (<30%)')

# Chart formatting.
plt.title('Recession Probability Forecast\nSeptember 2024 - February 2025',
          fontsize=14, pad=20)
plt.xlabel('Date', fontsize=12)
plt.ylabel('Probability', fontsize=12)

# Format the y-axis as a percentage.
plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda y, _: '{:.0%}'.for
mat(y)))
# Date rotation.
plt.xticks(rotation=45)
plt.legend(loc='upper right', title='Levels of risk')
```

```
# We add annotations with the exact values.
for idx, row in results_df.iterrows():
    plt.annotate(f'{row["Recession Probability"]:.1%}',
                 (idx, row["Recession Probability"]),
                 textcoords="offset points",
                 xytext=(0,10),
                 ha='center',
                 fontweight='bold')

# We set the borders of the graph.
plt.ylim(-0.05, 1.05)
# Layout optimization.
plt.tight_layout()
plt.show()
results_df
```




```
           Date  Recession Probability
141  2024-10-01                0.862716
142  2024-11-01                0.476150
143  2024-12-01                0.187531
144  2025-01-01                0.219110
145  2025-02-01                0.168936
146  2025-03-01                0.188394
```

```
# Probab. of recession => as arithmetic mean of probabilities for all months.
avg_probability = (results_df['Recession Probability'].mean() * 100).round(2)
f_probability = f"{avg_probability} %"
print(">>> ------ >> ----- > ----")
print(f"Average probability of a recession over the entire period: {f_probabili
ty}")
```

```
>>> ------ >> ----- > ----
Average probability of a recession over the entire period: 35.05 %
```

The **first method** works on the following principle:

- Takes the last 6 months of real data.
- Directly applies a trained model to this data to predict the probability of a recession for each of these months.
- It does not try to predict future feature values, but works with the last known values.
- cb_model.predict_proba(latest_data) calculates the probability of a recession occurring in the future based on the most recently observed economic indicators.
- Fictitious future dates for the next 6 months are generated so that the model can predict the probability of a recession for each of them.

The **second method** (with ARIMA) is probably less reliable because:

- Includes two stages of prediction, which increases the potential error.
- ARIMA predictions 6 months ahead can be inaccurate, especially in volatile economic conditions.
- Errors from ARIMA predictions are carried over to the final model.

The **first method is more reliable** because:

- Works directly with real data.
- There is only one stage of prediction.
- Less likely to accumulate errors.
- A more conservative approach that relies on current economic indicators.

It is possible to use both methods in parallel and compare the results, as each of them can cover different aspects of the economic situation.

4.3 Third Way of Forecasting.

It uses a **Rolling Window Approach** that:

The **rolling_window_prediction** function uses a fixed-size rolling window (window_size=24 months) that moves through the data.

For each iteration:

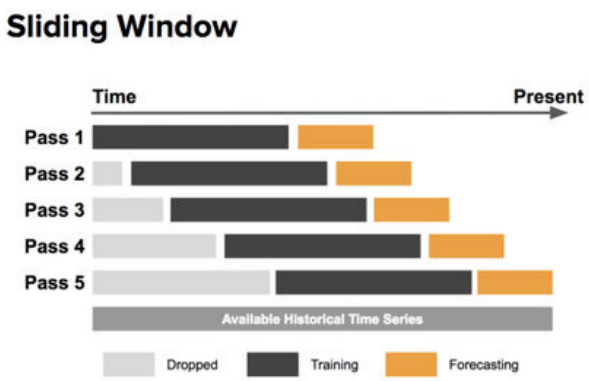
- Takes a 24-month window for training (train_window).
- Uses the next 6 months (prediction_horizon) for forecasting (future_window).
- Moves the window forward one step.

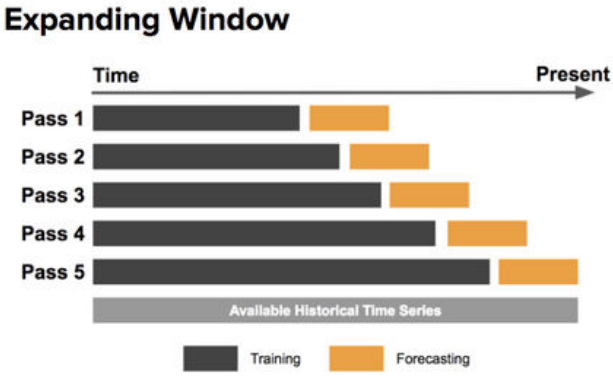
Advantages of this approach:

- Accounts for changes in the relationships between variables over time.
- More realistic than using a single fixed model for the entire period.
- Allows the model to adapt to new trends in the data.
- This approach uses the sliding window with forgetting method.
- Takes a fixed number of observations (window_size=24).
- At each iteration, the old data is "forgotten" and replaced with new ones.
- The window "slides" forward, maintaining a constant length.

Differences in approaches:

- **Sliding window** (current approach): More adaptive to changes, gives more weight to recent events and is more suitable for variable time dependencies.
- **Expanding window**: Uses the entire available history, more stable for recurring patterns and better for consistent time dependencies.





The main **disadvantages** of the sliding window with forgetting method:

Data problems:

- Loses valuable historical information about rare events.
- May miss important long-term cycles and trends.
- With a small window, it may "forget" important crisis periods.
- Risks ignoring recurring patterns from the past.

Statistical limitations:

- Smaller training sample (only within the window).
- Larger variation in estimates due to smaller sample size.
- Potentially more unstable predictions.
- Increased risk of overfitting in a small window.

Technical challenges:

- Sensitivity to the choice of window size.
- Difficulties in determining the optimal window length.
- With a window that is too small - excessive reactivity to noise.
- With a window that is too large - loss of the advantages of the method.

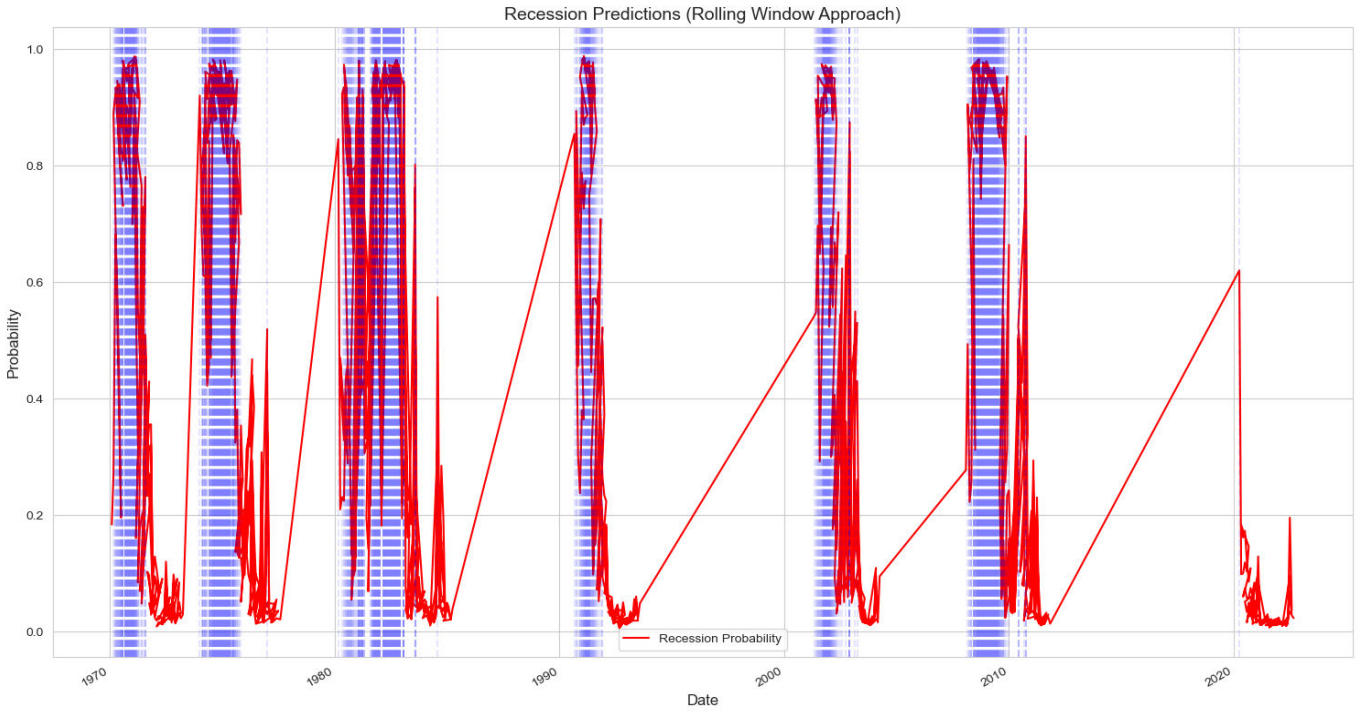
Problems with rare events (such as recessions):

- May miss rare events in the window entirely.
- Insufficient examples for training the model.
- Risk of missing early warning signals.
- Tendency to underestimate the probability of rare events.

```
def rolling_window_prediction(df, features, target, model, window_size=24, pred
iction_horizon=6):
    """
    Rolling window approach for recession prediction.

    Parameters:
    df : pandas DataFrame
        Input data with features and target variable.
    features : list
        List of feature names.
    target : str
        Target variable (e.g., 'recession').
    model : sklearn-like model
        Model to be trained and used for predictions.
    window_size : int
        Size of the rolling window for training (in months).
```

```
    prediction_horizon : int
        Number of months to predict ahead.
    Returns:
    pandas DataFrame
        Results with dates, predicted probabilities, and risk levels.
    """
    predictions = []
    probabilities = []
    dates = []

    for i in range(len(df) - window_size - prediction_horizon + 1):
        # Define training and future windows.
        train_window = df.iloc[i:i + window_size]
        future_window = df.iloc[i + window_size:i + window_size + prediction_ho
rizon]

        # Train model on the rolling window.
        X_train = train_window[features]
        y_train = train_window[target]

        # Skip windows where the target has only one unique value.
        if y_train.nunique() <= 1:
            continue

        model.fit(X_train, y_train)

        # Predict on future window.
        X_future = future_window[features]
        pred = model.predict(X_future)
        prob = model.predict_proba(X_future)[:, 1]

        predictions.extend(pred)
        probabilities.extend(prob)
        dates.extend(future_window.index)

    # Create a results DataFrame.
    results = pd.DataFrame({
        'date': dates,
        'predicted_recession': predictions,
        'recession_probability': probabilities
    }).set_index('date')

    # Add risk levels.
    results['risk_level'] = pd.cut(results['recession_probability'],
                                   bins=[0, 0.3, 0.6, 1],
                                   labels=['Low', 'Medium', 'High'])
    return results

# Load and prepare data.
data = pd.read_csv('data/collected_data.csv').set_index('date')
data.index = pd.to_datetime(data.index)

# Define features and target.
features = [col for col in data.columns if col != 'recession']
```

```
target = 'recession'

# Create the model.
cboost_model = CatBoostClassifier(
    auto_class_weights='Balanced',
    iterations=200,
    depth=8,
    learning_rate=0.05,
    random_state=42,
    verbose=False
)

# Use rolling window prediction.
results = rolling_window_prediction(data, features, target, cboost_model, windo
w_size=24, prediction_horizon=6)

# Visualize results
fig, ax = plt.subplots(figsize=(15, 8))
results['recession_probability'].plot(ax=ax, label='Recession Probability', col
or='red')
results[results['predicted_recession'] == 1].index.map(
    lambda x: ax.axvline(x=x, color='blue', linestyle='--', alpha=0.1, label='P
redicted Recession' if x == results.index[0] else "")
)
ax.set_title('Recession Predictions (Rolling Window Approach)', fontsize=14)
ax.set_xlabel('Date', fontsize=12)
ax.set_ylabel('Probability', fontsize=12)
ax.legend()
plt.tight_layout()
plt.show()
```



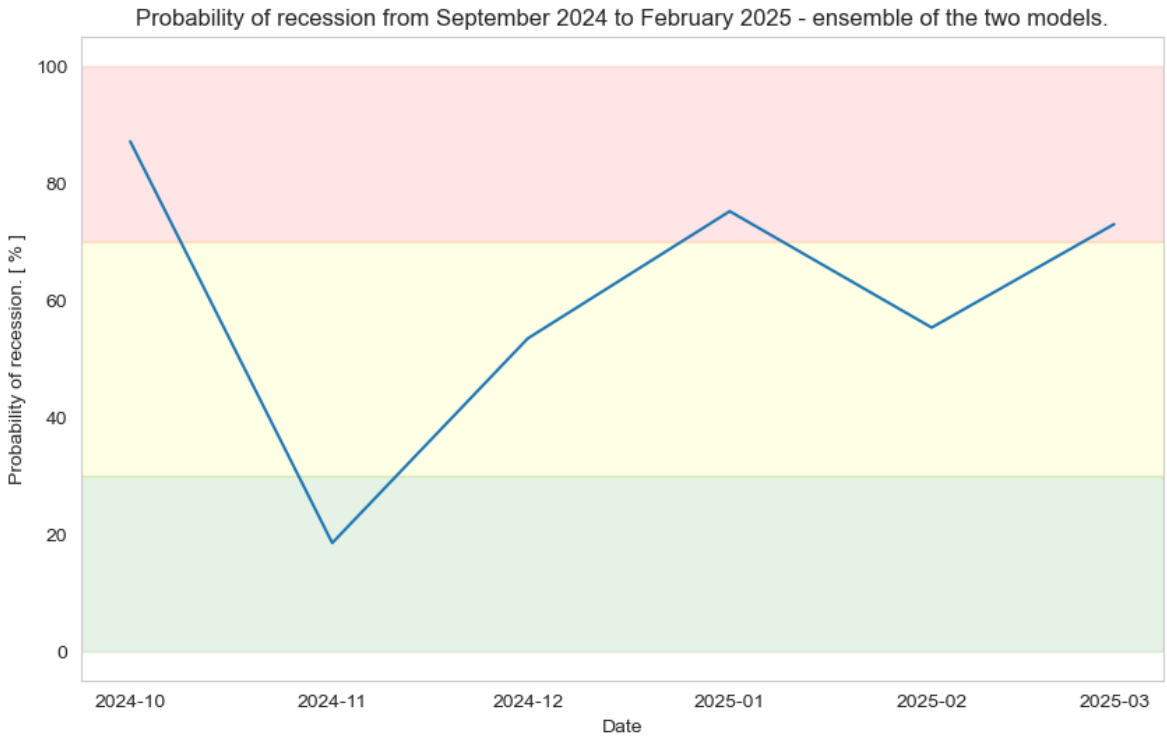
After window_size=24, the result that is obtained is almost independent of its size, measured in months. The **Rolling Window Approach** method for predicting economic recession **works, but if we have classical economics**. Which does not evolve from "bright to brighter future" with the help of massive money

infusions and thus avoiding recessions. When recognized economic recessions over large periods are very small or absent, then the model apparently does not do well. That is why **it is not included in the final stage** of the study.

4.4 Final lines.

We create **an ensemble of the two methods** and obtain the summary below. Because of the possible errors of the second method, we give it a weight of 0.3. The first method receive a weight of 0.7. It does not claim to be faithful, it is simply **an attempt to summarize the results** of the entire study.

```
# Parameters for the new DataFrame. Monthly index.
dates = pd.date_range(start='2024-10-01', end='2025-03-01', freq='MS')

# Calculating the probability of a recession by taking only the last 6 values.
recession_probabilities = (
    future_prob_df['probability_of_recession'].iloc[-6:].values * 0.7 * 100 +
    results_df['Recession Probability'].iloc[-6:].values * 0.3 * 100
)

# Create the new DataFrame.
new_df = pd.DataFrame(data=recession_probabilities, index=dates, columns=['Rece
ssion Probability'])

# Create a graph of the probability of a recession.
plt.figure(figsize=(10, 6))
plt.plot(new_df.index, new_df['Recession Probability'])
plt.title('Probability of recession from September 2024 to February 2025 - ense
mble of the two models.')

# We are adding color zones for the different risk levels.
plt.axhspan(70, 100, color='red', alpha=0.1, label='High risk (>70%)')
plt.axhspan(30, 70, color='yellow', alpha=0.1, label='Medium risk (30-70%)')
plt.axhspan(0, 30, color='green', alpha=0.1, label='Low risk (<30%)')

plt.xlabel('Date')
plt.ylabel('Probability of recession. [ % ]')
plt.grid()
plt.show()

new_df
```




```
            Recession Probability
2024-10-01              87.054083
2024-11-01              18.467138
2024-12-01              53.396982
2025-01-01              75.128673
2025-02-01              55.264570
2025-03-01              72.924928
```

```
# Probab. of recession => as arithmetic mean of probabilities for all months.
avg_probability = (new_df['Recession Probability'].mean()).round(2)
f_probability = f"{avg_probability} %"
print(">>> ------ >> ----- > ----")
print("Average recession probability of the ensemble of the three methods over
the entire period: ", f_probability)
```

```
>>> ------ >> ----- > ----
Average recession probability of the ensemble of the three methods over the ent
ire period:  60.37 %
```

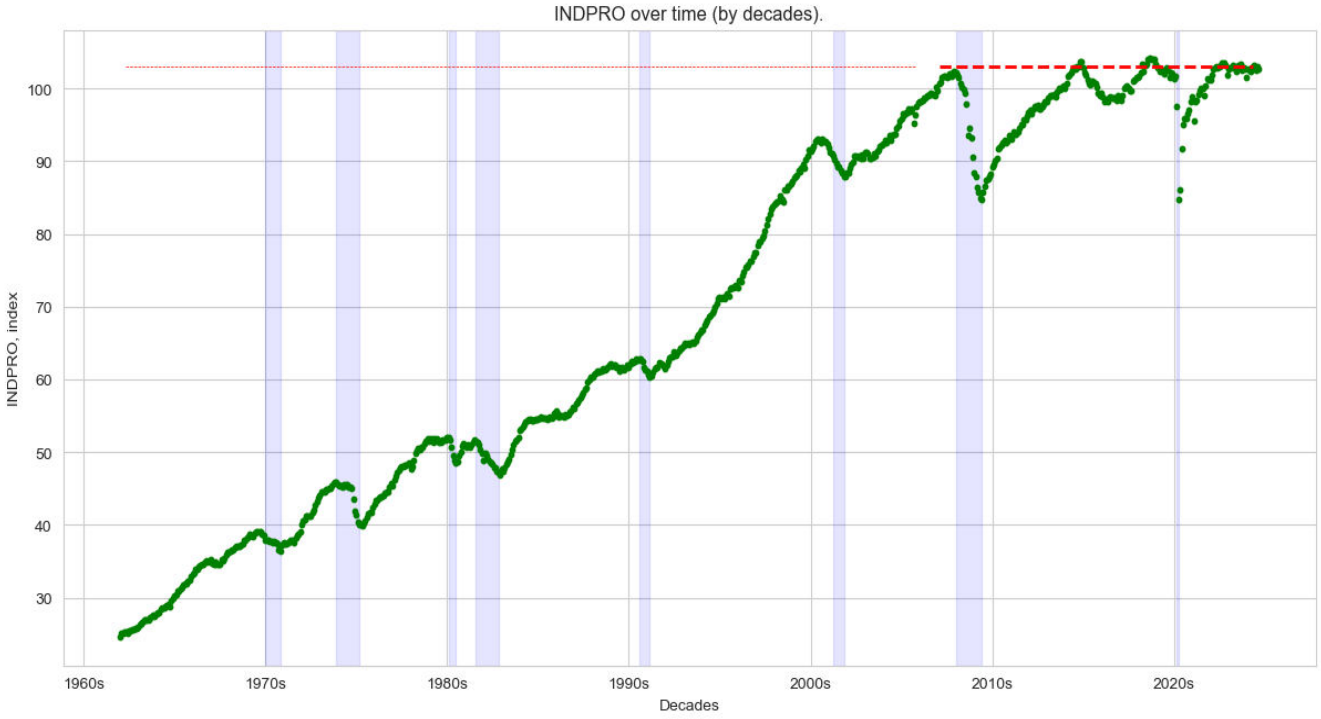
Cute polar bear Cute polar bear saying: **Hi**! On financial markets bears are usually not that cute... (free photo by Hans-Jurgen Mager)

5. Conclusions

- It makes **no sense to make a 3-month forecast** for an economic recession, as the collection of statistics is received with a delay of almost 2 months. The actual forecast that will be obtained will be for a month.
- A **one-year** forecast is also possible, but in the current tense geopolitical environment, this **loses any real meaning**. That's why we got this - a 6 month forecast.
- It is not appropriate to make a recession prediction based on an ensemble of the three forecasting methods. Their "logic" at work is too different. But in all three models, in the general case, the immediate probability of a recession is medium or high. Moments with low probability predictions are very few.
- The **average probability of a recession** in the period under study (October 2024 - March 2025) is **over 60%**. The summation of the three methods of obtaining the probability of a recession is a kind of Zoom OUT :)
- If only the **first method** turns out to be **historically accurate**, then we have no right to such averaging, even with coefficients. (The facts later confirmed exactly that.)
- The big brothers from STATISTA are probably right, predicting a **61.79%** chance of a recession in **August 2025**. [14] Well, the current study gives you about the same, but **without** the annual **fee** of 12 * 149 USD = 1788 USD. But with this difference that you see the entire process of predicting a future recession.
- **When will a recession really hit?** Elementary Watson! This has long since become a **political** rather than an economic **decision**. They will declare a recession when it is convenient for them. Or inevitably, as was the case with COVID-19. The US **recession that started in 2008 has never ended**! Why is this not apparent? Because of the "helicopter money"! If you don't know what this term means: Google it. I visualize the proof of the thesis below:

```
create_recession_chart()
# We make sure that the 'date' column is of type datetime.
indpro['date'] = pd.to_datetime(indpro['date'])
```

```
plt.scatter(indpro['date'], indpro['indpro'], s=10, color='green')
# Readability settings.
plt.ylabel('INDPRO, index')
plt.title('INDPRO over time (by decades).')
# Add a horizontal line.
plt.axhline(y=103, color='r', linestyle='--', xmin = 0.05, xmax = 0.68, linewid
th=0.5)
plt.axhline(y=103, color='r', linestyle='--', xmin = 0.70, xmax = 0.95, linewid
th=2)
plt.show()
```



After the recession in 2008, we don't have an increase in industrial production, do we? We are not talking about the so-called "service economy".

- How long can the effect of "helicopter money" (US Debt Clock) last? According to economists dealing with the subject, until the debts reach 200% of the Gross Domestic Product (GDP). In a bad geopolitical situation up to 175%. [15] But this is already the subject of further research...
- Still, will they declare a recession? Let's see. Before a recession, the poor stock up on salt, sugar, canned goods. Bankers stock up on... **gold**! I.e. before a recession or depression, the price of gold jumps a lot! On 23 October 2024, the price of gold reached an **all-time high** in real and nominal terms. [16] Even the peak around the 1980s recession, adjusted for inflation, is now smaller than the last one. I leave the conclusions to you...

Resources:

- **Recession Prediction using Machine Learning** https://towardsdatascience.com/recession-prediction-using-machine-learning-de6eee16ca94
- **Real-time Sahm Rule Recession Indicator** https://fred.stlouisfed.org/series/SAHMREALTIME
- **Industrial Production: Total Index** https://fred.stlouisfed.org/series/INDPRO
- **Historical data: S&P 500 - U.S.** https://stooq.com/q/d/?s=%5Espx&c=0&d1=19620101&d2=20241001&i=m

- **Market Yield on U.S. Treasury Securities at 10-Year Constant Maturity** https://fred.stlouisfed.org/series/dgs10
- **10-Year Treasury Constant Maturity Minus Federal Funds Rate** https://fred.stlouisfed.org/series/T10YFF
- **Monthly Unemployment Rate** https://fred.stlouisfed.org/series/UNRATE
- **Personal Consumption Expenditures: Chain-type Price Index** https://fred.stlouisfed.org/series/PCEPI
- **Total Nonfarm Payroll** https://fred.stlouisfed.org/series/PAYEMS
- **New Privately-Owned Housing Units Started: Total Units** https://fred.stlouisfed.org/series/HOUST
- **NBER based Recession Indicators for the United States** https://fred.stlouisfed.org/series/USREC
- **Why we have to remove highly correlated features in Machine Learning?** https://medium.com/@sujathamudadla1213/why-we-have-to-remove-highly-correlated-features-in-machine-learning-9a8416286f18
- **Autoregressive integrated moving average** https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average#External_links
- **Projected monthly probability of a recession in the United States from August 2020 to August 2025** https://www.statista.com/statistics/1239080/us-monthly-projected-recession-probability/
- **Bloomberg: US stuck in debt denial** https://opposition.bg/bloomberg-sasht-zamryaha-v-sastoyanie-na-otritsanie-na-dalga/
- **Gold reached its highest price in history** https://tavex.bg/zlatoto-dostigna-naj-visokata-si-cena-v-istoriyata/

Chapter 4: **Analysis of the US Economy – part 3**

```
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
from datetime import datetime
from scipy import stats
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.model_selection import KFold
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import Adam
import warnings
import random
import os

from datetime import timedelta
from statsmodels.tsa.api import VAR

# Set random seeds for reproducibility.
SEED = 42
np.random.seed(SEED)
tf.random.set_seed(SEED)
random.seed(SEED)
os.environ['PYTHONHASHSEED'] = str(SEED)
# Ignore warnings.
warnings.filterwarnings('ignore')
```

Date: June 2025, Deep Learning project

Abstract:

In the third part of this study, we answer the questions: what are the prospects for the US economy and under what conditions would the debt-to-GDP ratio reach 175%. This percentage is considered a critical point beyond which the sustainability of the economy is already in question. Several additional trends are also outlined, relevant conclusions are drawn.

1. Introduction

This study is a continuation of the previous two. The latter predicted the likelihood of a recession and indicated why one might not be declared. From here we ask ourselves the next logical question: can we measure the resilience of the US economy? And how? And also, how does US debt fit into all these processes of economic sustainability?

On the site Sustainable Economic Indicators we read: **Total GDP** and **GDP per capita** are good indicators of the relative strengths of the worlds nations. Other indicators may include:

- Investment in public, business and private assets (headline);

- Social investment;
- Rate of inflation;
- Government borrowing and debt;
- Competitiveness/productivity;
- Trade/exports/imports.

However, it is increasingly being recognised these traditional economic indicators alone do not provide an adequate measure of an economy's sustainability. To improve upon these indicators, we need to know how resource efficient our economy is, both in terms of the natural resources that it consumes (and wastes) and its utilisation of labour capital. Some indicators that have been proposed by the Government in this respect include:

- Waste production, and energy and water consumption;
- Transport indicators;
- Education and employment;
- Consumer expenditure;
- Environmental management and reporting. [1]

This study **cannot cover all aspects of the sustainability of an economy**, from internal social contradictions... to the state of the environment (ecology). But we **can safely use the most important indicators, which we can put into numbers** and then with **AI models**:

- **GROSS DOMESTIC PRODUCT** (**GDP**) => Competitiveness/productivity;
- **Real GDP per capita** (A939RX0Q048SBEA) => Competitiveness/productivity;
- **Total Public Debt as Percent of GDP** (GFDEGDQ188S) => Competitiveness/productivity;
- **Consumer Price Index** (CPI) => Consumer expenditure;
- **Industrial Production Index** (INDPRO) => Competitiveness/productivity;
- **S&P 500 Index** (SP500) => Investment in public, business and private assets;
- **Total Nonfarm Payroll** (PAYEMS) => Competitiveness/productivity (jobs);
- **Unemployment** (UNRATE) => Education and employment;
- **Federal Funds Effective Rate** (FEDFUNDS) => Rate of inflation;
- **Federal Debt: Total Public Debt** (GFDEBTN) => Government borrowing and debt;
- **Personal Saving Rate** (PSAVERT) => Social investment;
- **Trade Balance: Goods and Services, Balance of Payments Basis** (BOPGSTB) => Trade/exports/imports;
- **Total Debt Securities; Liability** (ASTDSL) => Government borrowing and debt;
- Federal government current expenditures: **Interest payments** (A091RC1Q027SBEA) => Government borrowing and debt;
- **Federal government current tax receipts** (W006RC1Q027SBEA) => Competitiveness/productivity;
- **Slope of the yield curve** (T10YFF) => Government borrowing and debt;
- **Federal Government: Current Expenditures** (FGEXPND) => Social investment;

For some of the data we will be able to easily interpolate back in time. Some of the data that we can include are not freely available, and others are either abandoned for collection back in time, or are being collected recently. **Back in time**, in the general case, we get a practical limit from **January 1966**, and for the moment we only have a limited amount of this data, which is published quarter by quarter. And with a great delay.

2. Collection of economic indicators

2.1 GROSS DOMESTIC PRODUCT (GDP)

From this site FEDERAL RESERVE BANK of ST. LOUIS we only download information for the period of our study: **Gross Domestic Product (GDP)**. [2] Units: **Billions of Dollars**, Seasonally Adjusted. Frequency: Quarterly.

```
gdp = pd.read_csv('data/GDP.csv') # Read the CSV file.
gdp.columns = ["date", "gdp"]
gdp['gdp'] = gdp['gdp'] * 1_000_000_000 # Convert to dollars.
```

Data for the period under study are given by **quartals**, not by months. To obtain all the requested data from **01.1962** to **the end of the study period**, we use a **linear approximation**.

```
gdp['date'] = pd.to_datetime(gdp['date'])
gdp_approx = gdp.copy()

# Create a new DF with all dates from 1966-01-01 to end of period: 2025-03-01.
# Latest real data: 2025-01-01.
all_dates = pd.date_range(start='1966-01-01', end='2025-03-01', freq='MS')
gdp_all_dates = pd.DataFrame({'date': all_dates})

# Merge the two DataFrames and interpolate.
gdp_approx = pd.merge(gdp_all_dates, gdp, on='date', how='left')

# Linear interpolation for existing data.
gdp_approx['gdp'] = gdp_approx['gdp'].interpolate(method='linear')

# Find the last real date and extrapolate beyond it.
last_real_date = gdp['date'].max()
extrapolation_mask = gdp_approx['date'] > last_real_date

if extrapolation_mask.any():
    # Calculate average monthly change from last 12 months.
    last_12_months = gdp_approx[gdp_approx['date'] <= last_real_date].tail(12)
    monthly_changes = last_12_months['gdp'].diff().dropna()
    avg_monthly_change = monthly_changes.mean()

    # Get the last real value.
    last_real_value = gdp_approx[gdp_approx['date'] <= last_real_date]['gdp'].i
loc[-1]

    # Extrapolate for future dates.
    future_indices = gdp_approx[extrapolation_mask].index
    for i, idx in enumerate(future_indices):
        gdp_approx.loc[idx, 'gdp'] = last_real_value + avg_monthly_change * (i
+ 1)

# Remove the 'year' and 'month' columns.
```

```
if 'year' in gdp_approx.columns:
    gdp_approx = gdp_approx.drop(columns=['year'])
if 'month' in gdp_approx.columns:
    gdp_approx = gdp_approx.drop(columns=['month'])

del gdp
gdp_approx.tail()
```

```
          date           gdp
706 2024-11-01  2.980812e+13
707 2024-12-01  2.989238e+13
708 2025-01-01  2.997664e+13
709 2025-02-01  3.008770e+13
710 2025-03-01  3.019876e+13
```

2.2 CONSUMER PRICE INDEX (CPI)

From this site FEDERAL RESERVE BANK of ST. LOUIS we only download information for the period of our study: **Consumer Price Index (CPI)** for All Urban Consumers, All Items in U.S. City Average (CPIAUCSL). [3] Units: **Index**, Seasonally Adjusted. Frequency: Monthly.

```
cpi = pd.read_csv('data/CPIAUCSL.csv') # Read the CSV file.
cpi.columns = ["date", "cpi"]
cpi.tail()
```

```
           date      cpi
706  2024-11-01  316.449
707  2024-12-01  317.603
708  2025-01-01  319.086
709  2025-02-01  319.775
710  2025-03-01  319.615
```

2.3 Industrial Production Index (INDPRO)

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Industrial Production Index (INDPRO)** [4], Seasonally Adjusted. Frequency: Monthly.

```
indpro = pd.read_csv('data/INDPRO.csv') # Read the CSV file.
indpro.columns = ["date", "indpro"]
indpro.tail()
```

```
           date    indpro
706  2024-11-01  101.9503
707  2024-12-01  103.0723
708  2025-01-01  103.2131
709  2025-02-01  104.1490
710  2025-03-01  103.8865
```

2.4 S&P 500 Index (SP500)

From this site Stooq we download information for the period of our study: **S&P 500 Index (SP500)** [5], Frequency: Monthly. This is the only publicly available data for the S&P 500 Index, which contains information all the way back to 1962!

```
sp = pd.read_csv('data/SP500.csv') # Read the CSV file.
# Keep only the required columns.
```

```
keep_col = ['Date', 'Close']
sp500 = sp[keep_col]
del sp
# Renaming columns.
sp500.columns = ['date', 'sp500']
sp500.tail()
```

```
           date    sp500
706  2024-11-30  6032.38
707  2024-12-31  5881.63
708  2025-01-31  6040.53
709  2025-02-28  5954.50
710  2025-03-31  5611.85
```

2.5 Total Nonfarm Payroll (PAYEMS)

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Total Nonfarm Payroll (PAYEMS)** [6], Units: **Thousands of Persons**, Seasonally Adjusted. Frequency: Monthly.

```
payems = pd.read_csv('data/PAYEMS.csv') # Read the CSV file.
payems.columns = ["date", "payems"]
payems.tail()
```

```
           date  payems
706  2024-11-01  158619
707  2024-12-01  158942
708  2025-01-01  159053
709  2025-02-01  159155
710  2025-03-01  159275
```

2.6 UNEMPLOYMENT (UNRATE)

From this site FEDERAL RESERVE BANK of ST. LOUIS we only download information for the period of our study: **Monthly Unemployment Rate (UNRATE)**. [7] Units: **Percent**, Seasonally Adjusted. Frequency: Monthly.

```
unrate = pd.read_csv('data/UNRATE.csv') # Read the CSV file.
unrate.columns = ["date", "unrate"]
unrate["unrate"] = unrate["unrate"] / 100 # To get in percentages.
unrate.tail()
```

```
           date  unrate
706  2024-11-01   0.042
707  2024-12-01   0.041
708  2025-01-01   0.040
709  2025-02-01   0.041
710  2025-03-01   0.042
```

2.7 Federal Funds Effective Rate (FEDFUNDS)

From this site Federal Reserve Bank of St. Louis we only download information for the period of our study: **Federal Funds Effective Rate (FEDFUNDS)**, Units: **Percent**. Frequency: Monthly [8]

```
fedfunds = pd.read_csv('data/FEDFUNDS.csv') # Read the CSV file.
fedfunds.columns = ["date", "fedfunds"]
```

```
fedfunds["fedfunds"] = fedfunds["fedfunds"] / 100 # To get in percentages.
fedfunds.tail()
```

```
           date  fedfunds
706  2024-11-01    0.0464
707  2024-12-01    0.0448
708  2025-01-01    0.0433
709  2025-02-01    0.0433
710  2025-03-01    0.0433
```

2.8 Total Public Debt (GFDEBTN)

From this site Federal Reserve Bank of St. Louis we only download information for the period of our study: **Federal Debt: Total Public Debt (GFDEBTN)**, Units: **Millions of Dollars**, Not Seasonally Adjusted. Frequency: Quarterly [9]

```
debt = pd.read_csv('data/GFDEBTN.csv') # Loading data.
debt.columns = ["date", "debt"]
debt['debt'] = debt['debt'] * 1_000_000 # Convert to dollars.

# Convert column 'date' to datetime.
debt['date'] = pd.to_datetime(debt['date'])

# Create a new DF with all dates from 1966-01-01 to the end of the period.
all_dates = pd.date_range(start='1966-01-01', end='2025-03-01', freq='MS')
debt_all_dates = pd.DataFrame({'date': all_dates})

# Merging the two DataFrames and interpolation.
debt_approx = pd.merge(debt_all_dates, debt, on='date', how='left')

# Linear interpolation.
debt_approx['debt'] = debt_approx['debt'].interpolate(method='linear')

# Remove the 'year' and 'month' columns, if they exist.
if 'year' in debt_approx.columns:
    debt_approx = debt_approx.drop(columns=['year'])
if 'month' in debt_approx.columns:
    debt_approx = debt_approx.drop(columns=['month'])

del debt
debt_approx.tail(6)
```

```
          date          debt
705 2024-10-01  3.621860e+13
706 2024-11-01  3.621717e+13
707 2024-12-01  3.621574e+13
708 2025-01-01  3.621431e+13
709 2025-02-01  3.621431e+13
710 2025-03-01  3.621431e+13
```

The project is being written in June 2025, and the latest data is for January 2025. Unfortunately, the last quarter of 2024 has a symbolic drop in the data. But this year, the US debt is constantly increasing, so we leave the last 3 months the same - as the approximation calculated it.

2.9 Personal Saving Rate (PSAVERT)

From this site Federal Reserve Bank of St. Louis we only download information for the period of our study: **Personal Saving Rate (PSAVERT)**, Units: **Percent**, Seasonally Adjusted. Frequency: Monthly [10] Personal saving is equal to personal income less personal outlays and personal taxes; it may generally be viewed as the portion of personal income that is used either to provide funds to capital markets or to invest in real assets such as residences.

```
pasavert = pd.read_csv('data/PSAVERT.csv') # Loading data.
pasavert.columns = ["date", "pasavert"]
pasavert["pasavert"] = pasavert["pasavert"] / 100 # To get in percentages.
pasavert.tail()
```

```
           date  pasavert
706  2024-11-01     0.039
707  2024-12-01     0.035
708  2025-01-01     0.041
709  2025-02-01     0.044
710  2025-03-01     0.043
```

2.10 Real gross domestic product per capita (A939RX0Q048SBEA)

From this site FEDERAL RESERVE BANK of ST. LOUIS we only download information for the period of our study: **Real gross domestic product per capita (A939RX0Q048SBEA)**. [11] Units: **Dollars**, Seasonally Adjusted. Frequency: Quarterly.

```
gdp_pca = pd.read_csv('data/A939RX0Q048SBEA.csv') # Read the CSV file.
gdp_pca.columns = ["date", "gdp_pca"]
```

Data for the period under study are given by **quartals**, not by months. To obtain all the requested data from **01.1966** to **the end of the study period**, we use a **linear approximation**.

```
gdp_pca['date'] = pd.to_datetime(gdp_pca['date'])
gdp_pca_approx = gdp_pca.copy()

# Create a new DF with all dates from 1966-01-01 to end of period: 2025-03-01.
# Latest real data: 2025-01-01.
all_dates = pd.date_range(start='1966-01-01', end='2025-03-01', freq='MS')
gdp_pca_all_dates = pd.DataFrame({'date': all_dates})

# Merge the two DataFrames and interpolate.
gdp_pca_approx = pd.merge(gdp_pca_all_dates, gdp_pca, on='date', how='left')

# Linear interpolation for existing data.
gdp_pca_approx['gdp_pca'] = gdp_pca_approx['gdp_pca'].interpolate(method='linea
r')

# Find the last real date and extrapolate beyond it.
last_real_date = gdp_pca['date'].max()
extrapolation_mask = gdp_pca_approx['date'] > last_real_date

if extrapolation_mask.any():
    # Calculate average monthly change from last 12 months.
    last_12_months = gdp_pca_approx[gdp_pca_approx['date'] <= last_real_date].t
ail(12)
    monthly_changes = last_12_months['gdp_pca'].diff().dropna()
```

```
    avg_monthly_change = monthly_changes.mean()

    # Get the last real value.
    last_real_value = gdp_pca_approx[gdp_pca_approx['date'] <= last_real_date][
'gdp_pca'].iloc[-1]

    # Extrapolate for future dates.
    future_indices = gdp_pca_approx[extrapolation_mask].index
    for i, idx in enumerate(future_indices):
        gdp_pca_approx.loc[idx, 'gdp_pca'] = last_real_value + avg_monthly_chan
ge * (i + 1)

# Remove the 'year' and 'month' columns.
if 'year' in gdp_pca_approx.columns:
    gdp_pca_approx = gdp_pca_approx.drop(columns=['year'])
if 'month' in gdp_pca_approx.columns:
    gdp_pca_approx = gdp_pca_approx.drop(columns=['month'])

del gdp_pca
gdp_pca_approx.tail()
```

```
          date       gdp_pca
706 2024-11-01  68963.333333
707 2024-12-01  68920.666667
708 2025-01-01  68878.000000
709 2025-02-01  68949.272727
710 2025-03-01  69020.545455
```

2.11 Total Public Debt as Percent of Gross Domestic Product (GFDEGDQ188S)

From this site FEDERAL RESERVE BANK of ST. LOUIS we can download information for the period of our study: **Total Public Debt as Percent of Gross Domestic Product (GFDEGDQ188S)**. [12] Units: **Percent of GDP**, Seasonally Adjusted. Frequency: Quarterly. Here we have a **problem**. Of all the economic indicators, this one is updated the latest, i.e. **has the largest time lag**! However, we **can simply calculate it** based on the data already collected:

```
# We calculate: Total Public Debt as Percent of Gross Domestic Product.
debt_per_gdp = pd.DataFrame() # Creating a new DataFrame.
# Copy dates.
debt_per_gdp['date'] = gdp_approx['date']

# Calculating the debt-to-GDP ratio.
debt_per_gdp['debt_per_gdp'] = (debt_approx['debt'] / gdp_approx['gdp'])
debt_per_gdp.tail()
```

```
          date  debt_per_gdp
706 2024-11-01      1.215010
707 2024-12-01      1.211538
708 2025-01-01      1.208084
709 2025-02-01      1.203625
710 2025-03-01      1.199198
```

2.12 Trade Balance: Goods and Services, Balance of Payments Basis (BOPGSTB)

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Trade Balance: Goods and Services, Balance of Payments Basis (BOPGSTB)**. Units: **Millions of Dollars**, Seasonally Adjusted. Frequency: Quarterly. Monthly [13] Here we have a problem. The information has only been collected since January 1992. Looking at the graph of this indicator from the site United States Balance of Trade [14], we can apply interpolation to generate data back to January 1966. Because the data is not freely available, but is paid for. US Balance of Trade

```
tb_file = pd.read_csv('data/BOPGSTB.csv') # Loading data.
tb_file.columns = ["date", "tb"]
tb_file['tb'] = tb_file['tb'] * 1_000_000 # Convert to dollars.
```

The last date with data is January 1992. Using CLAUDE.ai we approximate the missing data to January 1966. The missing values are not large in absolute value and the error will be many times smaller than if we use linear approximation. We receive from **Claude.ai** a file in **.csv** format with dates from **January 1966** to **December 1991**. We load and merge the data for the trade balance into one file.


```
approx_file = pd.read_csv('data/TB_approx.csv') # Loading data.
approx_file.columns = ["date", "tb"]
approx_file['tb'] = approx_file['tb'] * 1_000_000 # Convert to dollars.

# Merge the two DataFrames.
tb = pd.concat([approx_file, tb_file], ignore_index=True)

# Sort by date.
tb = tb.sort_values('date')
# Remove any duplicate entries.
tb = tb.drop_duplicates(subset=['date'], keep='last')

tb.tail()
```

```
           date            tb
706  2024-11-01 -7.975200e+10
707  2024-12-01 -9.694800e+10
708  2025-01-01 -1.302730e+11
709  2025-02-01 -1.220110e+11
710  2025-03-01 -1.383160e+11
```

2.13 Total Debt Securities; Liability (ASTDSL):

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Total Debt Securities; Liability (ASTDSL)**. [19] Units: **Millions of Dollars**, Not Seasonally Adjusted, Frequency: Quarterly. **ASTDSL** represents the total level of debt securities issued by all sectors of the U.S. economy. This includes debt instruments, such as bonds, corporate debt securities, and other forms of debt, which are obligations of various economic agents, including the government, corporations, and households.

```
astdsl = pd.read_csv('data/ASTDSL.csv') # Read the CSV file.
astdsl.columns = ["date", "astdsl"]
astdsl['astdsl'] = astdsl['astdsl'] * 1_000_000 # Convert to dollars.

astdsl['date'] = pd.to_datetime(astdsl['date'])
astdsl_approx = astdsl.copy()

# Create a new DF with all dates from 1966-01-01 to end of period: 2025-03-01.
# Latest real data: 2025-01-01.
all_dates = pd.date_range(start='1966-01-01', end='2025-03-01', freq='MS')
astdsl_all_dates = pd.DataFrame({'date': all_dates})

# Merge the two DataFrames and interpolate.
astdsl_approx = pd.merge(astdsl_all_dates, astdsl, on='date', how='left')

# Linear interpolation for existing data.
astdsl_approx['astdsl'] = astdsl_approx['astdsl'].interpolate(method='linear')

# Find the last real date and extrapolate beyond it.
last_real_date = astdsl['date'].max()
extrapolation_mask = astdsl_approx['date'] > last_real_date

if extrapolation_mask.any():
    # Calculate average monthly change from last 12 months.
    last_12_months = astdsl_approx[astdsl_approx['date'] <= last_real_date].tai
l(12)
    monthly_changes = last_12_months['astdsl'].diff().dropna()
    avg_monthly_change = monthly_changes.mean()

    # Get the last real value.
    last_real_value = astdsl_approx[astdsl_approx['date'] <= last_real_date]['a
stdsl'].iloc[-1]

    # Extrapolate for future dates.
    future_indices = astdsl_approx[extrapolation_mask].index
    for i, idx in enumerate(future_indices):
        astdsl_approx.loc[idx, 'astdsl'] = last_real_value + avg_monthly_change
* (i + 1)
```

```
# Remove the 'year' and 'month' columns.
if 'year' in astdsl_approx.columns:
    astdsl_approx = astdsl_approx.drop(columns=['year'])
if 'month' in astdsl_approx.columns:
    astdsl_approx = astdsl_approx.drop(columns=['month'])

del astdsl
astdsl_approx.tail()
```

	date	astdsl
706	2024-11-01	6.211803e+13
707	2024-12-01	6.238642e+13
708	2025-01-01	6.265482e+13
709	2025-02-01	6.288636e+13
710	2025-03-01	6.311791e+13

2.14 Federal government current expenditures: Interest payments (A091RC1Q027SBEA)

From this site Federal Reserve Bank of St. Louis we only download information for the period of our study: **Federal government current expenditures: Interest payments (A091RC1Q027SBEA)**, Units: **Billions of Dollars**, Seasonally Adjusted. Frequency: Quarterly [16]

```
interest = pd.read_csv('data/A091RC1Q027SBEA.csv') # Read the CSV file.
interest.columns = ["date", "interest"]
interest['interest'] = interest['interest'] * 1_000_000_000 # Convert to dollar
s.

# Convert column 'date' to datetime.
interest['date'] = pd.to_datetime(interest['date'])
interest_approx = interest.copy()

# Create a new DF with all dates from 1966-01-01 to end of period: 2025-03-01.
# Latest real data: 2025-01-01.
all_dates = pd.date_range(start='1966-01-01', end='2025-03-01', freq='MS')
interest_all_dates = pd.DataFrame({'date': all_dates})

# Merge the two DataFrames and interpolate.
interest_approx = pd.merge(interest_all_dates, interest, on='date', how='left')

# Linear interpolation for existing data.
interest_approx['interest'] = interest_approx['interest'].interpolate(method='l
inear')

# Find the last real date and extrapolate beyond it.
last_real_date = interest['date'].max()
extrapolation_mask = interest_approx['date'] > last_real_date

if extrapolation_mask.any():
    # Calculate average monthly change from last 12 months.
    last_12_months = interest_approx[interest_approx['date'] <= last_real_date]
.tail(12)
    monthly_changes = last_12_months['interest'].diff().dropna()
    avg_monthly_change = monthly_changes.mean()
```

```
    # Get the last real value.
    last_real_value = interest_approx[interest_approx['date'] <= last_real_date
][['interest'].iloc[-1]

    # Extrapolate for future dates.
    future_indices = interest_approx[extrapolation_mask].index
    for i, idx in enumerate(future_indices):
        interest_approx.loc[idx, 'interest'] = last_real_value + avg_monthly_ch
ange * (i + 1)

# Remove the 'year' and 'month' columns.
if 'year' in interest_approx.columns:
    interest_approx = interest_approx.drop(columns=['year'])
if 'month' in interest_approx.columns:
    interest_approx = interest_approx.drop(columns=['month'])

del interest
interest_approx.tail()
```

```
          date      interest
706 2024-11-01  1.121040e+12
707 2024-12-01  1.117675e+12
708 2025-01-01  1.114311e+12
709 2025-02-01  1.117691e+12
710 2025-03-01  1.121070e+12
```

2.15 Federal government current tax receipts (W006RC1Q027SBEA)

From this site Federal Reserve Bank of St. Louis we only download information for the period of our study: **Federal government current tax receipts (W006RC1Q027SBEA)**, Units: **Billions of Dollars**, Seasonally Adjusted. Frequency: Quarterly [17]

```
tax = pd.read_csv('data/W006RC1Q027SBEA.csv') # Read the CSV file.
tax.columns = ["date", "tax"]
tax['tax'] = tax['tax'] * 1_000_000_000 # Convert to dollars.

# Convert column 'date' to datetime.
tax['date'] = pd.to_datetime(tax['date'])
tax_approx = tax.copy()

# Create a new DF with all dates from 1966-01-01 to end of period: 2025-03-01.
# Latest real data: 2025-01-01.
all_dates = pd.date_range(start='1966-01-01', end='2025-03-01', freq='MS')
tax_all_dates = pd.DataFrame({'date': all_dates})

# Merge the two DataFrames and interpolate.
tax_approx = pd.merge(tax_all_dates, tax, on='date', how='left')

# Linear interpolation for existing data.
tax_approx['tax'] = tax_approx['tax'].interpolate(method='linear')

# Find the last real date and extrapolate beyond it.
last_real_date = tax['date'].max()
extrapolation_mask = tax_approx['date'] > last_real_date
```

```
if extrapolation_mask.any():
    # Calculate average monthly change from last 12 months.
    last_12_months = tax_approx[tax_approx['date'] <= last_real_date].tail(12)
    monthly_changes = last_12_months['tax'].diff().dropna()
    avg_monthly_change = monthly_changes.mean()

    # Get the last real value.
    last_real_value = tax_approx[tax_approx['date'] <= last_real_date]['tax'].i
loc[-1]

    # Extrapolate for future dates.
    future_indices = tax_approx[extrapolation_mask].index
    for i, idx in enumerate(future_indices):
        tax_approx.loc[idx, 'tax'] = last_real_value + avg_monthly_change * (i
+ 1)

# Remove the 'year' and 'month' columns.
if 'year' in tax_approx.columns:
    tax_approx = tax_approx.drop(columns=['year'])
if 'month' in tax_approx.columns:
    tax_approx = tax_approx.drop(columns=['month'])

del tax
tax_approx.tail()
```

```
          date           tax
706 2024-11-01  3.203521e+12
707 2024-12-01  3.227351e+12
708 2025-01-01  3.251182e+12
709 2025-02-01  3.269819e+12
710 2025-03-01  3.288457e+12
```

2.16 Slope of the yield curve (T10YFF):

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Slope of the yield curve (T10YFF)**. [18] Units: **Percent**, Seasonally Adjusted, Frequency: Daily.

```
t10yff = pd.read_csv('data/T10YFF.csv') # Read the CSV file.
t10yff.columns = ["date", "t10yff"]
# Replace '.' with NaN and convert column 't10yff' to numeric type.
t10yff['t10yff'] = pd.to_numeric(t10yff['t10yff'], errors='coerce')

# Group by month and calculate the arithmetic mean for each month.
t10yff['date'] = pd.to_datetime(t10yff['date'])
t10yff.set_index('date', inplace=True)
t10yff_avg = t10yff.resample('MS').mean()

# Create a new DF with the first number of the month and the arithmetic mean.
t10yff_avg = t10yff_avg.reset_index()
t10yff_avg.tail()
```

```
          date    t10yff
706 2024-11-01 -0.290000
707 2024-12-01 -0.093333
708 2025-01-01  0.299048
```

```
709 2025-02-01  0.121053
710 2025-03-01 -0.049524
```

2.17 Federal Government: Current Expenditures (FGEXPND):

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Federal Government: Current Expenditures (FGEXPND)**. [19] Units: **Billions of Dollars**, Seasonally Adjusted, Frequency: Quarterly.

```
fgexpnd = pd.read_csv('data/FGEXPND.csv') # Read the CSV file.
fgexpnd.columns = ["date", "fgexpnd"]
fgexpnd['fgexpnd'] = fgexpnd['fgexpnd'] * 1_000_000_000 # Convert to dollars.

# Convert column 'date' to datetime.
fgexpnd['date'] = pd.to_datetime(fgexpnd['date'])
fgexpnd_approx = fgexpnd.copy()

# Create a new DF with all dates from 1966-01-01 to end of period: 2025-03-01.
# Latest real data: 2025-01-01.
all_dates = pd.date_range(start='1966-01-01', end='2025-03-01', freq='MS')
fgexpnd_all_dates = pd.DataFrame({'date': all_dates})

# Merge the two DataFrames and interpolate.
fgexpnd_approx = pd.merge(fgexpnd_all_dates, fgexpnd, on='date', how='left')

# Linear interpolation for existing data.
fgexpnd_approx['fgexpnd'] = fgexpnd_approx['fgexpnd'].interpolate(method='linea
r')

# Find the last real date and extrapolate beyond it.
last_real_date = fgexpnd['date'].max()
extrapolation_mask = fgexpnd_approx['date'] > last_real_date

if extrapolation_mask.any():
    # Calculate average monthly change from last 12 months.
    last_12_months = fgexpnd_approx[fgexpnd_approx['date'] <= last_real_date].t
ail(12)
    monthly_changes = last_12_months['fgexpnd'].diff().dropna()
    avg_monthly_change = monthly_changes.mean()

    # Get the last real value.
    last_real_value = fgexpnd_approx[fgexpnd_approx['date'] <= last_real_date][
'fgexpnd'].iloc[-1]

    # Extrapolate for future dates.
    future_indices = fgexpnd_approx[extrapolation_mask].index
    for i, idx in enumerate(future_indices):
        fgexpnd_approx.loc[idx, 'fgexpnd'] = last_real_value + avg_monthly_chan
ge * (i + 1)

# Remove the 'year' and 'month' columns.
if 'year' in fgexpnd_approx.columns:
    fgexpnd_approx = fgexpnd_approx.drop(columns=['year'])
if 'month' in fgexpnd_approx.columns:
    fgexpnd_approx = fgexpnd_approx.drop(columns=['month'])
```

```
del fgexpnd
fgexpnd_approx.tail()
```

```
          date       fgexpnd
706 2024-11-01  7.129362e+12
707 2024-12-01  7.148873e+12
708 2025-01-01  7.168383e+12
709 2025-02-01  7.201562e+12
710 2025-03-01  7.234741e+12
```

2.18 Preprocessing data

```
# Create a new empty table.
data = pd.DataFrame(columns=[
    'gdp',            # GROSS DOMESTIC PRODUCT
    'cpi',            # Consumer price index
    'indpro',         # Industrial Production Index
    'sp500',          # S&P 500 Index
    'payems',         # Total Nonfarm Payroll
    'unrate',         # Unemployment
    'fedfunds',       # Federal Funds Effective Rate
    'debt',           # Total Public Debt
    'pasavert',       #  Personal Saving Rate
    'gdp_pca',        # Real GDP per capita
    'debt_per_gdp',   # Total Public Debt as Percent of GDP
    'tb',             # Trade Balance
    'astdsl',         # Total Debt Securities; Liability
    'interest',       # Federal government interest payments
    'tax',            # Federal government current tax receipts
    'fgexpnd',        # Federal Government Current Expenditures
    't10yff'          # Slope of the yield curve
])
# Add data to the columns.
data['gdp'] = gdp_approx['gdp']                 # _ USD
data['cpi'] = cpi['cpi']                        # _____ Index
data['indpro'] = indpro['indpro']               # _____ Index
data['sp500'] = sp500['sp500']                  # _____ Index
data['payems'] = payems['payems']               # _____ Persons
data['unrate'] = unrate['unrate']               # _____ Percent
data['fedfunds'] = fedfunds['fedfunds']         # _____ Percent
data['debt'] = debt_approx['debt']              # _ USD
data['pasavert'] = pasavert['pasavert']         # _____ Percent
data['gdp_pca'] = gdp_pca_approx['gdp_pca']     # _ USD
data['debt_per_gdp'] = debt_per_gdp['debt_per_gdp'] # _ Percent
data['tb'] = tb['tb']                           # _ USD
data['astdsl'] = astdsl_approx['astdsl']        # _ USD
data['interest'] = interest_approx['interest'] # _ USD
data['tax'] = tax_approx['tax']                 # _ USD
data['fgexpnd'] = fgexpnd_approx['fgexpnd']     # _ USD
data['t10yff'] = t10yff_avg['t10yff']           # _____ Percent

# Create list of monthly dates.
dates = pd.date_range(start='01-1966', end='03-2025', freq='MS')
formatted_dates = pd.to_datetime(dates)
```

```
# Assuming collected_data is already an existing DataFrame with some data.
# Check the length of data to ensure it matches the length of formatted_dates.
if len(data) == len(formatted_dates):
    data['date'] = formatted_dates
else:
    print("The length of collected_data does not match the length of formatted_
dates!")
data.set_index('date', inplace=True)
data.to_csv(r'data/data.csv') # We save the dataframe to a file.
data
```

```
                    gdp      cpi    indpro    sp500  payems  unrate  \
date
1966-01-01  7.957340e+11   31.880   33.1709    92.88   62529   0.040
1966-02-01  7.988163e+11   32.080   33.3859    91.22   62796   0.038
1966-03-01  8.018987e+11   32.180   33.8429    89.23   63192   0.038
1966-04-01  8.049810e+11   32.280   33.8967    91.06   63437   0.038
1966-05-01  8.098667e+11   32.350   34.2192    86.13   63712   0.039
...                  ...      ...       ...      ...     ...     ...
2024-11-01  2.980812e+13  316.449  101.9503  6032.38  158619   0.042
2024-12-01  2.989238e+13  317.603  103.0723  5881.63  158942   0.041
2025-01-01  2.997664e+13  319.086  103.2131  6040.53  159053   0.040
2025-02-01  3.008770e+13  319.775  104.1490  5954.50  159155   0.041
2025-03-01  3.019876e+13  319.615  103.8865  5611.85  159275   0.042

            fedfunds          debt  pasavert       gdp_pca  debt_per_gdp  \
date
1966-01-01    0.0442  3.209990e+11     0.112  24172.000000      0.403400
1966-02-01    0.0460  3.193650e+11     0.110  24178.333333      0.399798
1966-03-01    0.0466  3.177310e+11     0.107  24184.666667      0.396223
1966-04-01    0.0467  3.160970e+11     0.103  24191.000000      0.392676
1966-05-01    0.0490  3.189807e+11     0.112  24234.333333      0.393868
...              ...           ...       ...           ...           ...
2024-11-01    0.0464  3.621717e+13     0.039  68963.333333      1.215010
2024-12-01    0.0448  3.621574e+13     0.035  68920.666667      1.211538
2025-01-01    0.0433  3.621431e+13     0.041  68878.000000      1.208084
2025-02-01    0.0433  3.621431e+13     0.044  68949.272727      1.203625
2025-03-01    0.0433  3.621431e+13     0.043  69020.545455      1.199198

                      tb        astdsl      interest           tax  \
date
1966-01-01 -3.500000e+06  4.689650e+11  1.944000e+10  9.869200e+10
1966-02-01 -3.600000e+06  4.699753e+11  1.962933e+10  1.001187e+11
1966-03-01 -3.700000e+06  4.709857e+11  1.981867e+10  1.015453e+11
1966-04-01 -3.800000e+06  4.719960e+11  2.000800e+10  1.029720e+11
1966-05-01 -3.900000e+06  4.745483e+11  2.016800e+10  1.034133e+11
...                  ...           ...           ...           ...
2024-11-01 -7.975200e+10  6.211803e+13  1.121040e+12  3.203521e+12
2024-12-01 -9.694800e+10  6.238642e+13  1.117675e+12  3.227351e+12
2025-01-01 -1.302730e+11  6.265482e+13  1.114311e+12  3.251182e+12
2025-02-01 -1.220110e+11  6.288636e+13  1.117691e+12  3.269819e+12
2025-03-01 -1.383160e+11  6.311791e+13  1.121070e+12  3.288457e+12
```

```
                fgexpnd    t10yff
date
1966-01-01  1.360560e+11  0.288095
1966-02-01  1.381690e+11  0.242632
1966-03-01  1.402820e+11  0.229565
1966-04-01  1.423950e+11  0.111500
1966-05-01  1.439157e+11 -0.057143
...                  ...       ...
2024-11-01  7.129362e+12 -0.290000
2024-12-01  7.148873e+12 -0.093333
2025-01-01  7.168383e+12  0.299048
2025-02-01  7.201562e+12  0.121053
2025-03-01  7.234741e+12 -0.049524

[711 rows x 17 columns]
```

3. Analysis of economic sustainability

3.1 First simulation:

Input: 17 features. **Target**: Composite Health Score (we will create it as a weighted sum of normalized metrics). **Model**: Hybrid 1D CNN + LSTM architecture **Objective**: Assessing overall economic health by forecasting a composite index based on economic indicators and time series. Main functionalities of the model:

1. Creating a composite economic health index: The index is based on weighted values of economic indicators. Positive and negative metrics have a total weight of 1 and -1, respectively, which ensures a balanced measurement of economic conditions.
2. Using LSTM for temporal dependencies: Two LSTM layers allow capturing short-term and long-term dependencies in economic data.
3. Application of Dropout to prevent overfitting: After each LSTM layer, a Dropout layer with a coefficient of 0.3 is included to reduce the risk of overfitting.
4. Preparation of time series data: The data is prepared as sequences using the prepare_sequences function to make it suitable for time series training.
5. Data Normalization: Input and output data are normalized using MinMaxScaler to improve training stability.
6. Results Visualization:
   - Training Process: Plots of loss and MAE (mean absolute error) metrics.
   - Predictions: Plots for comparison between actual and predicted values, noting the standard deviation around the predictions.

Model Architecture:

The model combines 1D Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) layers to take advantage of the advantages of both technologies:

- **Conv1D** layer: Captures local temporal dependencies in sequences (kernel_size=5, 128 filters).
- **MaxPooling1D** layer: Reduces the dimensionality of the output from the Conv1D layer.
- First **LSTM** layer: Captures short-term and intermediate dependencies (256 units, return_sequences=True).

- **Dropout** layer: To prevent overfitting.
- Second **LSTM** layer: Summarizes the sequence into a single output vector (128 units, return_sequences=False).
- **Dropout** layer: Reduces the risk of overfitting.
- **Dense** layers: 64 units (ReLU activation) for additional processing. 1 unit for the final prediction (economic health).

Detailed description of the functionalities:

CNN part: 1D Convolution is applied to the time series to extract key local features from the input data. This is useful for recognizing patterns related to economic indicators.

LSTM part: Captures both short-term and long-term dependencies in economic data. The first LSTM layer preserves full consistency, while the second summarizes all the information into a single vector.

Composite index: The index combines positive and negative indicators with normalized weights to provide an objective measure of economic health.

Predictions: The model predicts future values of economic health based on consistent data from the past.

Advantages of the approach:

- The hybrid architecture (CNN + LSTM) combines spatial and temporal analysis.
- Normalization ensures stability of the training.
- Dropout layers prevent overfitting, even with complex architectures.
- The composite index offers an intuitive and interpretable measurement of economic health.

Here is a justification for the chosen weights in the new model:

Positive indicators (total = 1.0):

- GDP (0.15): Main indicator of economic growth. Lower coefficient than the maximum 0.15, because sometimes GDP can grow without reflecting real welfare.
- INDPRO (0.15): Industrial production directly indicates economic potential. Equally important with GDP, because it reflects real production activity.
- GDP_PCA (0.15): Per capita income: a direct indicator of the standard of living. Equal to GDP and industrial production due to the direct link to welfare.
- PAYEMS (0.10): Employment is a important indicator of economic stability. As much as GDP and INDPRO, because it directly affects social dynamics.
- PASAVERT (0.10): Savings indicate financial stability of households, but with less weight because they are a more indirect indicator.
- SP500 (0.10): The stock market index reflects market sentiment, but is more volatile and subjective.
- TAX (0.10): Tax revenues indicate economic activity, but are a more productive indicator.
- INTEREST (0.10): Interest payments show financial dynamics, but are a more peripheral indicator.
- T10YFF (0.05): The slope of the yield curve is an important, but most nuanced signal.

Negative indicators (total = -1.0):

- DEBT (-0.25): Public debt is the most critical risk, therefore it has the largest negative weight.
- CPI (-0.15): Inflation directly destroys economic stability, the second most important negative factor.
- TB (-0.10): The trade balance shows external economic health, significant, but not so critical.
- UNRATE (-0.10): Unemployment is an important social indicator.

- FEDFUNDS (-0.10): Interest rates affect borrowing and investment.
- DEBT_PER_GDP (-0.10): The debt/GDP ratio further describes the debt picture.
- FGEXPND (-0.10): Government spending indicates economic management.
- ASTDSL (-0.10): The total level of debt securities issued by all sectors of the U.S. economy.

The logic is to create a balanced composite index that:

- Captures different economic dimensions.
- Gives more weight to fundamental indicators.
- Balances positive and negative signals. The sum of all weights is zero, which provides a balanced model.
- Reflects the complexity of the economic system.

```
# Function to evaluate model performance using various metrics.
def model_evaluation(true_values, predictions, threshold=0):
    """
    Calculate regression metrics between true values and predictions

    Args:
        true_values: Array of actual/ground truth values
        predictions: Array of model predictions
        threshold: Value for binary classification (default: 0). Optional.

    Returns:
        Dictionary containing RMSE, MAE, and R2 score metrics
    """
    # Convert continuous values to binary based on threshold. Optional...
    # binary_predictions = (predictions > threshold).astype(int)
    # binary_true = (true_values > threshold).astype(int)

    # Calculate various regression metrics.
    metrics = {
        'rmse': np.sqrt(mean_squared_error(true_values, predictions)), # RMSE
        'mae': mean_absolute_error(true_values, predictions), # Mean Abs. Error
        'r2': r2_score(true_values, predictions)              # R-squared score
    }
    return metrics

# Function to visualize training metrics over epochs.
def plot_training_history(history):
    """
    Plot training and validation metrics history.

    Args:
        history: Keras history object containing training metrics.
    """
    plt.figure(figsize=(12, 4))

    # Plot loss metrics.
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.legend()
```

```
    # Plot MAE metrics.
    plt.subplot(1, 2, 2)
    plt.plot(history.history['mae'], label='Training MAE')
    plt.plot(history.history['val_mae'], label='Validation MAE')
    plt.title('Model MAE')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Function to visualize model predictions against actual values.
def plot_predictions(true_values, predictions, dates):
    """
    Create a comparison plot of predicted vs actual values with uncertainty.
    Args:
        true_values: Array of actual values
        predictions: Array of model predictions
        dates: Array of corresponding dates for x-axis
    """
    # Flatten arrays for plotting.
    true_values = np.array(true_values).flatten()
    predictions = np.array(predictions).flatten()

    # Create main plot.
    plt.figure(figsize=(15, 6))
    plt.plot(dates, true_values, label='Actual', alpha=0.7)
    plt.plot(dates, predictions, label='Predicted', alpha=0.7)

    # Add uncertainty band (±1 standard deviation).
    std_dev = predictions.std()
    plt.fill_between(
        dates,
        predictions - std_dev,
        predictions + std_dev,
        alpha=0.2,
        label='Prediction Std Dev'
    )

    # Add reference line at y=0
    plt.axhline(y=0, color='r', linestyle='-')
    # Customize plot appearance.
    plt.title('Predicted vs Actual Economic Health')
    plt.legend(loc='upper left')
    plt.tight_layout()
    plt.grid(True)
    plt.show()

# Economic health prediction class.
class EconomicHealthPredictor:
    def __init__(self, lookback=12):
        """
        Initialize predictor with specified lookback period.
        Args:
```

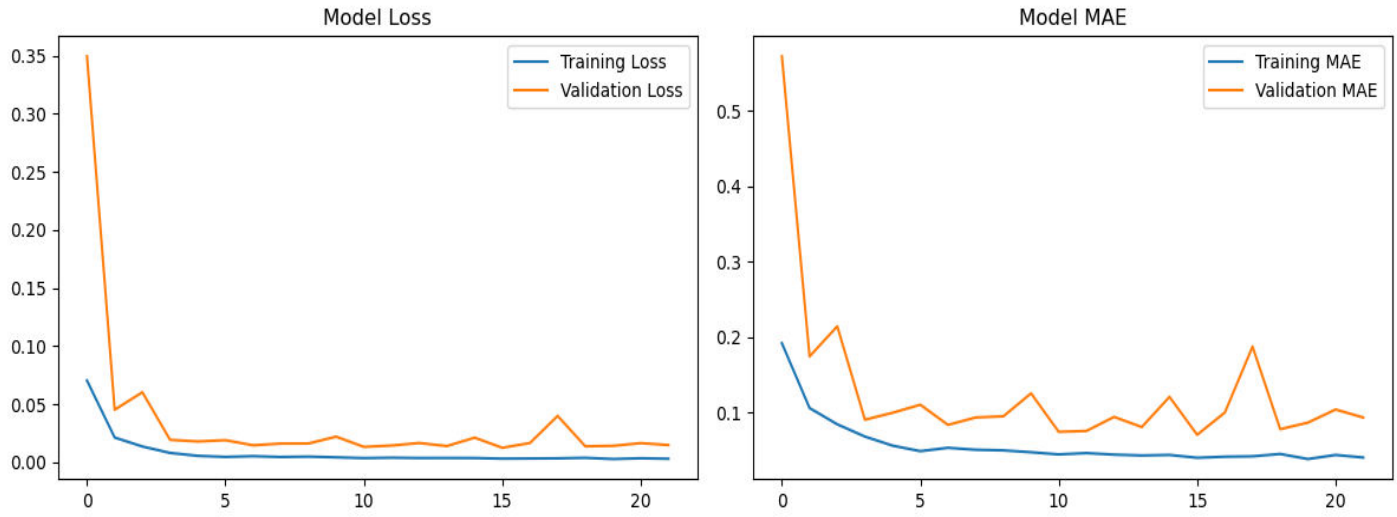
```
        lookback: Number of previous time steps to consider for prediction.
    """
    self.lookback = lookback
    self.scaler_X = MinMaxScaler()    # Scaler for input features.
    self.scaler_y = MinMaxScaler()    # Scaler for target variable.
    self.model = None       # NN model placeholder. The NN not yet created.

def create_composite_health_score(self, data):
    """
    Creates a composite index of economic health with updated indicators.

    Positive indicators (Sum = 1):
    - GDP (core economic growth indicator)
    - INDPRO (industrial production)
    - SP500 (market stability)
    - PAYEMS (employment)
    - GDP_PCA (population welfare)
    - PASAVERT (population savings)
    - TAX (tax revenues)
    - INTEREST (interest payments on government loans)
    - T10YFF (yield curve slope)

    Negative indicators (Sum = -1):
    - DEBT (total public debt)
    - CPI (inflation)
    - UNEMPLOYMENT (unemployment)
    - FEDFUNDS (cost of borrowing)
    - DEBT_PER_GDP (debt burden)
    - TB (trade deficit)
    - ASTDSL (total indebtedness)
    - FGEXPND (government spending)
    """
    weights = {
    # Positive indicators (total sum = 1).
        'gdp': 0.15,           # A key indicator of economic growth.
        'indpro': 0.15,        # Production capacity.
        'sp500': 0.1,          # Market confidence.
        'payems': 0.1,         # Employment.
        'gdp_pca': 0.15,       # Standard of living.
        'pasavert': 0.1,       # Household financial stability.
        'tax': 0.1,            # Tax revenues.
        'interest': 0.1,       # Interest payments.
        't10yff': 0.05,        # Yield curve slope.

    # Negative indicators (total sum = -1).
        'debt': -0.25,         # A major problem for the economy.
        'cpi': -0.15,          # Inflation.
        'unrate': -0.1,        # Social indicator.
        'fedfunds': -0.1,      # Impact on the cost of loans.
        'debt_per_gdp': -0.1, # Long-term financial stability.
        'tb': -0.1,            # Trade balance.
        'astdsl': -0.1,        # Energy dependence.
        'fgexpnd': -0.1        # Government spending.
```

```
        }
        # Normalize input data using MinMaxScaler.
        normalized_data = self.scaler_X.fit_transform(data)
        df_normalized = pd.DataFrame(normalized_data, columns=data.columns)

        # Calculate weighted sum of indicators.
        health_score = sum(
            df_normalized[col] * weight for col, weight in weights.items() if c
ol in df_normalized.columns
        )
        return health_score

    def prepare_sequences(self, X, y):
        """
        Create sequences of lookback periods for time series prediction.
        Args:
            X: Input features array.
            y: Target values array.

        Returns:
            Tuple of (X sequences, y sequences) for model training.
        """
        X_seq, y_seq = [], []
        for i in range(len(X) - self.lookback):
            X_seq.append(X[i:(i + self.lookback)])
            y_seq.append(y[i + self.lookback])
        return np.array(X_seq), np.array(y_seq)

    def build_model(self, input_shape):
        """
        Creates a model.
        """
        # Hybrid CNN-LSTM architecture for time series prediction.
        model = Sequential([
            Conv1D(128, kernel_size=5, activation='relu', input_shape=input_sha
pe),
            MaxPooling1D(pool_size=2),
            LSTM(256, return_sequences=True),
            Dropout(0.3),
            LSTM(128, return_sequences=False),
            Dropout(0.3),
            Dense(64, activation='relu'),
            Dense(1)
        ])
        model.compile(optimizer='adam', loss='mse', metrics=['mae'])
        return model

    def fit(self, X, epochs=22, validation_split=0.2, verbose=0):
        """
        Train the model on input data.
        Args:
            X: Input features DataFrame.
            epochs: Number of training epochs.
```

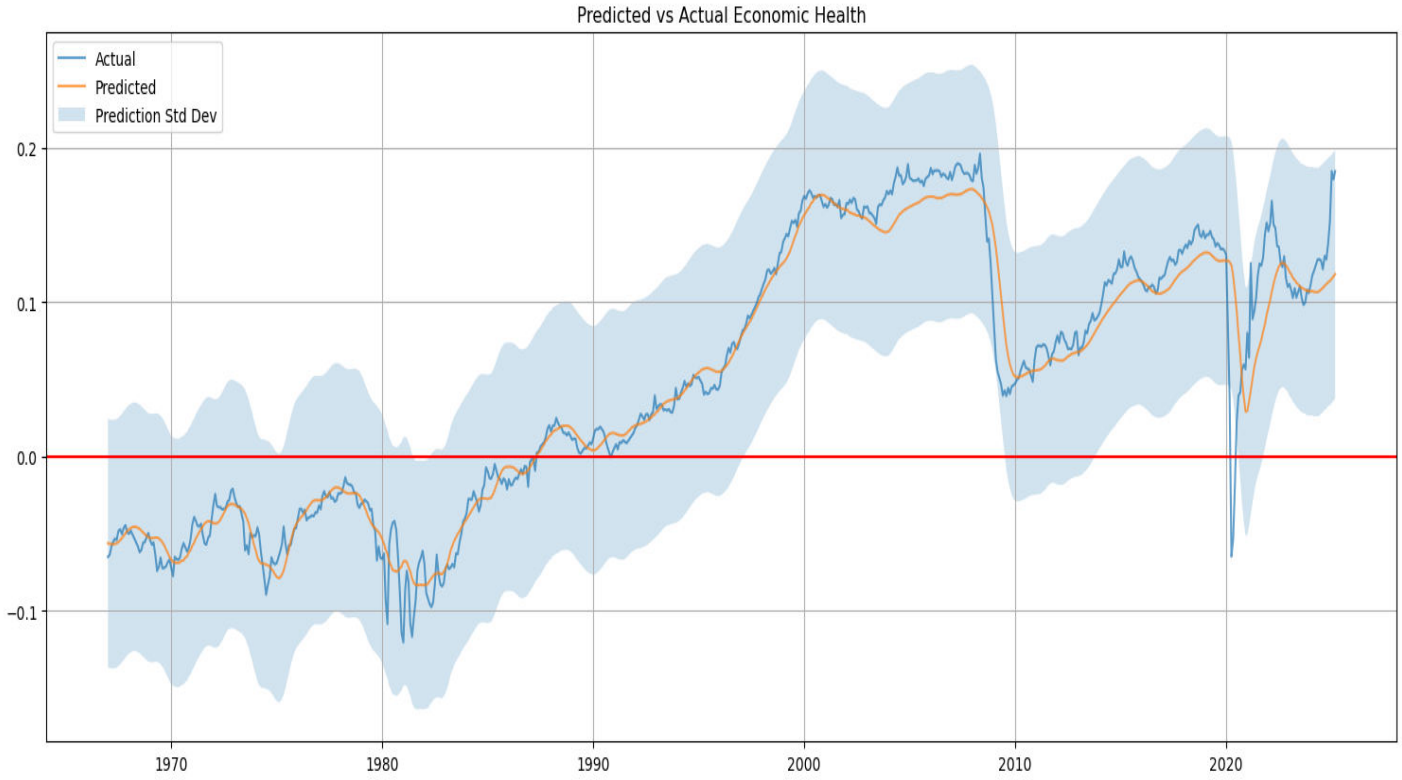
```
            validation_split: Fraction of data to use for validation.
            verbose: Verbosity level of training output.
        Returns:
            Training history.
        """
        # Calculate health score and scale data.
        y = self.create_composite_health_score(X)
        X_scaled = self.scaler_X.fit_transform(X)
        y_scaled = self.scaler_y.fit_transform(y.values.reshape(-1, 1))

        # Prepare sequences for time series prediction.
        X_seq, y_seq = self.prepare_sequences(X_scaled, y_scaled)
        self.model = self.build_model((self.lookback, X.shape[1]))

        # Add early stopping to prevent overfitting.
        early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore
_best_weights=True)

        history = self.model.fit(
            X_seq, y_seq,
            epochs=epochs,
            validation_split=validation_split,
            callbacks=[early_stopping],
            verbose=verbose
        )
        return history

    def predict(self, X):
        """
        Make predictions on new data.
        Args:
            X: Input features DataFrame.

        Returns:
            Array of predicted values in original scale.
        """
        X_scaled = self.scaler_X.transform(X)
        X_seq, _ = self.prepare_sequences(X_scaled, np.zeros(len(X_scaled)))
        predictions = self.model.predict(X_seq, verbose=0)
        return self.scaler_y.inverse_transform(predictions)

# Main analysis function.
def run_analysis(data):
    print("Training the model...")
    predictor = EconomicHealthPredictor(lookback=12)
    history = predictor.fit(data, epochs=22, validation_split=0.2, verbose=0)

    # Add back the training history plot.
    plot_training_history(history)

    # Making predictions.
    predictions = predictor.predict(data)
    health_score = predictor.create_composite_health_score(data)
```

```
    # Add model evaluation.
    metrics = model_evaluation(health_score[12:], predictions)

    # Print metrics.
    print("\nModel Eval. Metrics:", end=' ')
    print(", ".join(f"{metric}: {value:.3f}" for metric, value in metrics.items
()))
    print("\n")

    # Visualization of predictions.
    plot_predictions(health_score[12:], predictions, data.index[12:])

    return predictor, history, predictions, health_score, metrics

# Performing the analysis.
if __name__ == "__main__":
    # If we are loading data from a file.
    # data = pd.read_csv("data/data.csv", index_col=0, parse_dates=True)
    predictor = run_analysis(data)
```

```
Training the model...
```




```
Model Eval. Metrics: rmse: 0.020, mae: 0.012, r2: 0.947
```



Model Loss Loss is a measure of how well or poorly a model performs during training. It is a numerical value that shows the difference between the model's predictions and the true values. The Loss function depends on the type of task:

- For **regression**, **Mean Squared Error (MSE)** or **Mean Absolute Error (MAE)** is often used.
- For **classification**, **Cross-Entropy Loss** is often used. The plot of **Loss** usually shows the value of the loss versus the epochs.
- **The goal is for the loss to decrease with each epoch**, which means the model is learning and getting better. If the loss:
  - Decreases smoothly: the model is training well.
  - Stops decreasing or starts increasing: there may be **overfitting** (the model is adapting too much to the training data).
  - Does not decrease at all: the model may not learn (it is possible that the learning rate is too small or the architecture is not suitable).

Model MAE (Mean Absolute Error) Mean Absolute Error (MAE) is a metric that measures the average absolute difference between the predicted values and the true values.

- The formula for MAE is:

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

  where: $\mathbf{y_i}$ - are the true values, $\mathbf{y_i}$ hat - are the predicted values, **n** is the number of examples. Like Loss, MAE is also plotted against epochs.
- **The goal is for MAE to decrease over time**, indicating that the model is making more accurate predictions.
- MAE is easier to interpret than Loss because it is in the same units as the input data (for example, if you are predicting temperature in degrees, MAE will also be in degrees).

Are we done with this? Probably not. In the late 1970s, the US economy and the purchasing power of the dollar were in their heyday. And inflation over the decades does not stop working, which significantly

distorts the picture. That is, if at the beginning we have apples, at the end of the period we have pears. How do we compare which is bigger, apples or pears? **For an objective analysis, we need to exclude inflation** from the general picture. How? By multiplying the features that are measured in dollars by some factor that takes into account inflation.

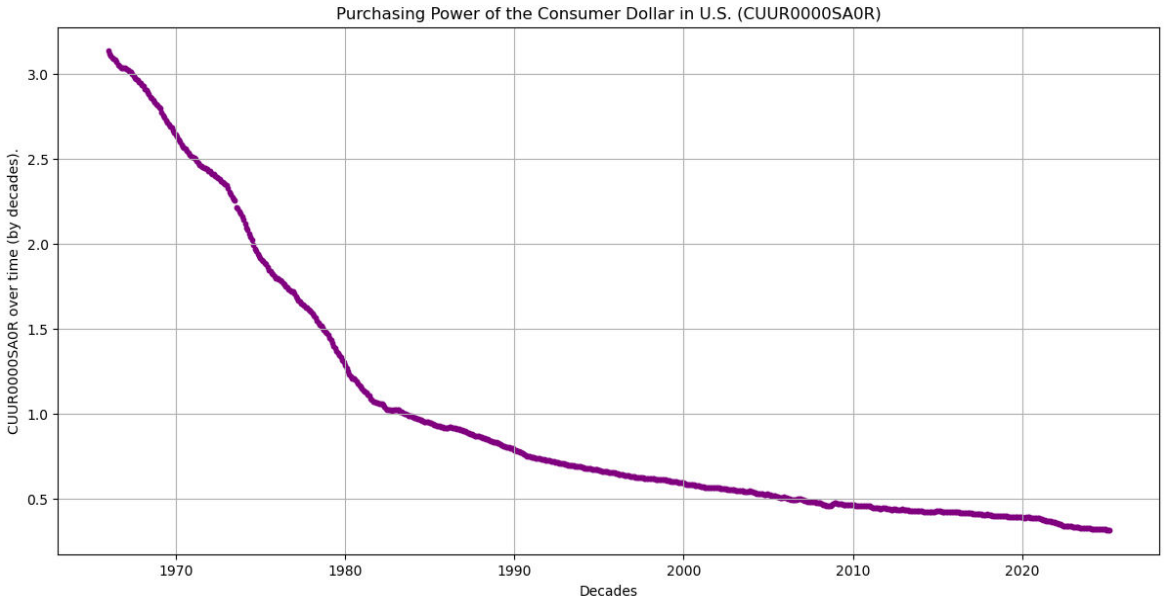
3.2 Second simulation:

From this site FEDERAL RESERVE BANK of ST. LOUIS we download information for the period of our study: **Purchasing Power of the Consumer Dollar in U.S. City Average (CUUR0000SA0R)**. [20] Units: **Index**, Frequency: Monthly.

```
deflator = pd.read_csv('data/CUUR0000SA0R.csv') # Loading data.
deflator.columns = ["date", "deflator"]

# Convert the % index to a number.
deflator['deflator'] = deflator['deflator'] / 100
deflator['date'] = pd.to_datetime(deflator['date'])
deflator.set_index('date', inplace=True)
deflator.tail()
```

```
            deflator
date
2024-11-01     0.317
2024-12-01     0.317
2025-01-01     0.315
2025-02-01     0.313
2025-03-01     0.313
```

```
# Create the chart.
plt.figure(figsize=(12, 6))
plt.xlabel('Decades')
plt.grid(True)
plt.tight_layout()
plt.scatter(deflator.index, deflator['deflator'], s=10, color='purple')
# Readability settings.
plt.ylabel('CUUR0000SA0R over time (by decades).')
plt.title('Purchasing Power of the Consumer Dollar in U.S. (CUUR0000SA0R)')
plt.show()
```



```
# Create a copy of the original DataFrame.
adjusted_data = data.copy()

# List of dollar columns to be adjusted.
usd_columns = ['gdp', 'debt', 'gdp_pca', 'tb', 'astdsl', 'interest', 'tax', 'fg
expnd']

# Multiply the values in the columns by the deflator.
for col in usd_columns:
    adjusted_data[col] = data[col] * deflator['deflator']
```

Once inflation is eliminated, another problem will arise. The economy will not go from boom to boom, from bright to even brighter future. And it will be almost impossible for the model to pass metrics with over 90% accuracy. It's tried and tested. More information describing anomalies in the economy could be added to help model efficiency. Two more features were chosen: recessions and the COVID-19 period. But this did not produce good results either. The problem was solved with **cross-validation** and adjustments to the model's hyperparameters. In addition, **months were removed** from the **COVID-19 period in such a way as not to interfere with the trend** in price movements. And why were they removed at all? Because of their **anti-market value**. At that time, businesses were being forced to close and "helicopter money" was being given out.

```
# Define a start and end date.
start_date = pd.Timestamp('2020-01-01')
end_date = pd.Timestamp('2021-11-01')

# List of all columns.
columns = ['gdp', 'cpi', 'indpro', 'sp500', 'payems', 'unrate', 'fedfunds',
           'debt', 'pasavert', 'gdp_pca', 'debt_per_gdp', 'tb', 'astdsl',
           'interest', 'tax', 'fgexpnd', 't10yff']

# Period selection mask.
period_mask = (adjusted_data.index >= start_date) & (adjusted_data.index <= end
_date)

# Creating a dataframe with the correct columns and data types.
data_dict = {col: pd.Series(dtype='float64') for col in columns}
interpolation_df = pd.DataFrame(data_dict, index=[start_date, end_date])

# Copy the values for the start and end date.
for col in columns:
    interpolation_df.loc[start_date, col] = float(adjusted_data.loc[start_date,
col])
    interpolation_df.loc[end_date, col] = float(adjusted_data.loc[end_date, col
])

# Add the intermediate dates and perform interpolation.
interpolation_df = interpolation_df.reindex(adjusted_data.loc[period_mask].inde
x)
interpolation_df = interpolation_df.interpolate(method='linear')

# Updating the original dataframe.
adjusted_data.loc[period_mask, columns] = interpolation_df[columns]
```

```
adjusted_data.to_csv(r'data/adjusted_data.csv') # We save the DF to a file.
adjusted_data # View the corrected DataFrame.
```

```
                    gdp      cpi    indpro    sp500    payems  unrate  \
date
1966-01-01  2.495422e+12   31.880   33.1709    92.88   62529.0   0.040
1966-02-01  2.490709e+12   32.080   33.3859    91.22   62796.0   0.038
1966-03-01  2.490697e+12   32.180   33.8429    89.23   63192.0   0.038
1966-04-01  2.491416e+12   32.280   33.8967    91.06   63437.0   0.038
1966-05-01  2.504108e+12   32.350   34.2192    86.13   63712.0   0.039
...                  ...      ...       ...      ...       ...     ...
2024-11-01  9.449175e+12  316.449  101.9503  6032.38  158619.0   0.042
2024-12-01  9.475884e+12  317.603  103.0723  5881.63  158942.0   0.041
2025-01-01  9.442641e+12  319.086  103.2131  6040.53  159053.0   0.040
2025-02-01  9.417450e+12  319.775  104.1490  5954.50  159155.0   0.041
2025-03-01  9.452213e+12  319.615  103.8865  5611.85  159275.0   0.042

            fedfunds          debt  pasavert       gdp_pca  debt_per_gdp  \
date
1966-01-01    0.0442  1.006653e+12     0.112  75803.392000      0.403400
1966-02-01    0.0460  9.957801e+11     0.110  75388.043333      0.399798
1966-03-01    0.0466  9.868725e+11     0.107  75117.574667      0.396223
1966-04-01    0.0467  9.783202e+11     0.103  74871.145000      0.392676
1966-05-01    0.0490  9.862882e+11     0.112  74932.558667      0.393868
...              ...           ...       ...           ...           ...
2024-11-01    0.0464  1.148084e+13     0.039  21861.376667      1.215010
2024-12-01    0.0448  1.148039e+13     0.035  21847.851333      1.211538
2025-01-01    0.0433  1.140751e+13     0.041  21696.570000      1.208084
2025-02-01    0.0433  1.133508e+13     0.044  21581.122364      1.203625
2025-03-01    0.0433  1.133508e+13     0.043  21603.430727      1.199198

                      tb        astdsl      interest           tax  \
date
1966-01-01 -1.097600e+07  1.470674e+12  6.096384e+10  3.094981e+11
1966-02-01 -1.122480e+07  1.465383e+12  6.120426e+10  3.121700e+11
1966-03-01 -1.149220e+07  1.462881e+12  6.155678e+10  3.153998e+11
1966-04-01 -1.176100e+07  1.460828e+12  6.192476e+10  3.186983e+11
1966-05-01 -1.205880e+07  1.467303e+12  6.235946e+10  3.197540e+11
...                  ...           ...           ...           ...
2024-11-01 -2.528138e+10  1.969142e+13  3.553696e+11  1.015516e+12
2024-12-01 -3.073252e+10  1.977650e+13  3.543031e+11  1.023070e+12
2025-01-01 -4.103600e+10  1.973627e+13  3.510080e+11  1.024122e+12
2025-02-01 -3.818944e+10  1.968343e+13  3.498371e+11  1.023453e+12
2025-03-01 -4.329291e+10  1.975591e+13  3.508949e+11  1.029287e+12

                fgexpnd    t10yff
date
1966-01-01  4.266716e+11  0.288095
1966-02-01  4.308109e+11  0.242632
1966-03-01  4.357159e+11  0.229565
1966-04-01  4.407125e+11  0.111500
1966-05-01  4.449872e+11 -0.057143
...                  ...       ...
```

```
2024-11-01  2.260008e+12 -0.290000
2024-12-01  2.266193e+12 -0.093333
2025-01-01  2.258041e+12  0.299048
2025-02-01  2.254089e+12  0.121053
2025-03-01  2.264474e+12 -0.049524

[711 rows x 17 columns]
```

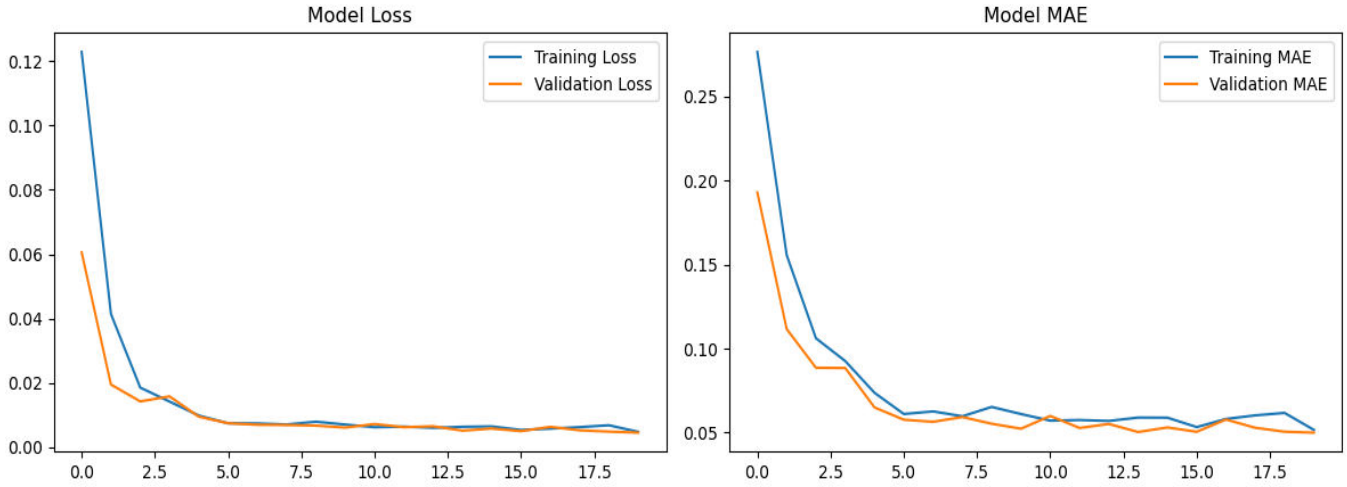
```python
class EconomicHealthPredictor:
    def __init__(self, lookback=12):
        """
        Initialize the Economic Health Predictor.
        Args:
            lookback: Number of time steps to look back for predictions.
        """
        self.lookback = lookback
        self.scaler_X = MinMaxScaler()    # Feature scaler.
        self.scaler_y = MinMaxScaler()    # Target variable scaler.
        self.model = None                 # Neural network model placeholder.

    @tf.function
    def _create_predict_function(self, inputs):
        return self.model(inputs, training=False)

    def create_composite_health_score(self, adjusted_data):
        """
        Calculate composite economic health score from multiple indicators.
        """
        weights = {
        # Positive indicators (total sum = 1).
            'gdp': 0.15,
            'indpro': 0.15,
            'sp500': 0.1,
            'payems': 0.1,
            'gdp_pca': 0.15,
            'pasavert': 0.1,
            'tax': 0.1,
            'interest': 0.1,
            't10yff': 0.05,

        # Negative indicators (total sum = -1).
            'debt': -0.25,
            'cpi': -0.15,
            'unrate': -0.1,
            'fedfunds': -0.1,
            'debt_per_gdp': -0.1,
            'tb': -0.1,
            'astdsl': -0.1,
            'fgexpnd': -0.1
       }
        # Scale the input data and calculate weighted health score.
        normalized_data = self.scaler_X.fit_transform(adjusted_data)
        df_normalized = pd.DataFrame(normalized_data, columns=adjusted_data.col
umns)
```

```
        health_score = sum(df_normalized[col] * weight for col, weight in weigh
ts.items() if col in df_normalized.columns)
        return health_score

    def prepare_sequences(self, X, y):
        """
        Prepare time series sequences for model training.
        Args:
            X: Input features array.
            y: Target values array.
        """
        X_seq, y_seq = [], []
        for i in range(len(X) - self.lookback):
            X_seq.append(X[i:(i + self.lookback)])
            y_seq.append(y[i + self.lookback])
        return np.array(X_seq), np.array(y_seq)

    def build_model(self, input_shape):
        """
        Creates a model.
        """
        # Deep CNN-LSTM hybrid architecture.
        model = Sequential([
            Conv1D(128, kernel_size=5, activation='relu', input_shape=input_sha
pe),
            MaxPooling1D(pool_size=2),
            LSTM(512, return_sequences=True),
            Dropout(0.4),
            LSTM(256, return_sequences=True),
            Dropout(0.3),
            LSTM(128, return_sequences=False),
            Dropout(0.2),
            Dense(64, activation='relu'),
            Dense(1)
        ])
        # Configure optimizer with custom learning rate.
        optimizer = Adam(learning_rate=1e-3)
        model.compile(optimizer=optimizer, loss='mse', metrics=['mae'])
        return model

    def fit(self, adjusted_data, epochs=20, validation_split=0.2, verbose=0):
        """
        Train the model on the provided data.
        Args:
            adjusted_data: Input features DataFrame.
            epochs: Number of training epochs.
            validation_split: Fraction of data for validation.
            verbose: Training output verbosity level.
        """
        # Prepare data for training.
        y = self.create_composite_health_score(adjusted_data)
        X_scaled = self.scaler_X.fit_transform(adjusted_data)
        y_scaled = self.scaler_y.fit_transform(y.values.reshape(-1, 1))
```

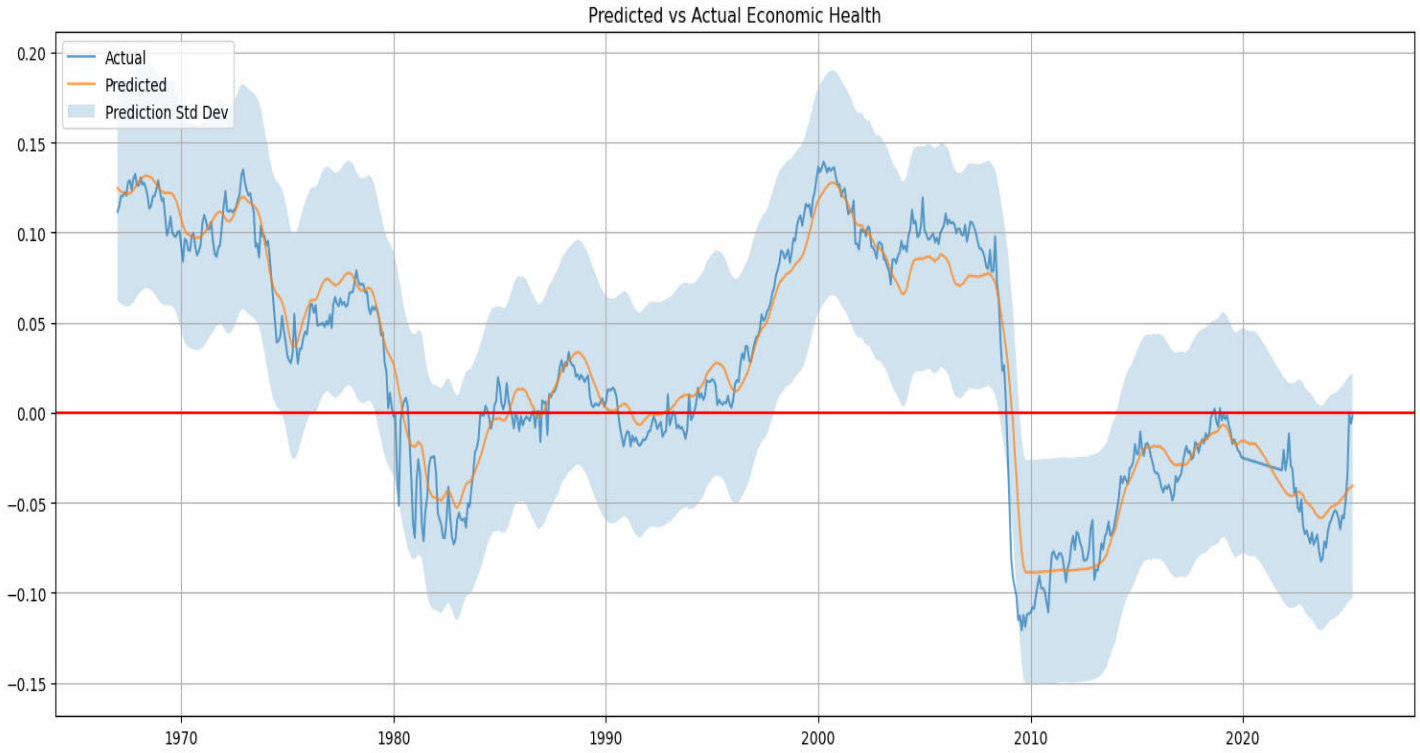
```
        X_seq, y_seq = self.prepare_sequences(X_scaled, y_scaled)

        # Initialize model and training settings.
        self.model = self.build_model((self.lookback, adjusted_data.shape[1]))
        early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore
_best_weights=True)

        history = self.model.fit(
            X_seq, y_seq,
            epochs=epochs,
            validation_split=validation_split,
            callbacks=[early_stopping],
            verbose=verbose,
            batch_size=32
        )
        return history

    def cross_validate(self, adjusted_data, epochs=20, k=5):
        """
        Perform k-fold cross-validation.
        Args:
            adjusted_data: Input features DataFrame.
            epochs: Number of training epochs per fold.
            k: Number of folds for cross-validation.
        Returns:
            Average metrics across folds and training history.
        """
        kf = KFold(n_splits=k, shuffle=True, random_state=42)
        metrics_list = []

        # Prepare data for cross-validation.
        y = self.create_composite_health_score(adjusted_data)
        X_scaled = self.scaler_X.fit_transform(adjusted_data)
        y_scaled = self.scaler_y.fit_transform(y.values.reshape(-1, 1))
        X_seq, y_seq = self.prepare_sequences(X_scaled, y_scaled)

        # Perform k-fold cross-validation.
        for train_index, test_index in kf.split(X_seq):
            X_train, X_test = X_seq[train_index], X_seq[test_index]
            y_train, y_test = y_seq[train_index], y_seq[test_index]

            self.model = self.build_model((self.lookback, adjusted_data.shape[1
]))
            history = self.model.fit(X_train, y_train, epochs=epochs, verbose=0
, batch_size=32, validation_data=(X_test, y_test))

            preds = self.model.predict(X_test)
            metrics = model_evaluation(y_test, preds)
            metrics_list.append(metrics)

        # Calculate average metrics across folds.
        avg_metrics = {key: np.mean([m[key] for m in metrics_list]) for key in
metrics_list[0]}
        return avg_metrics, history
```

```
    def predict(self, adjusted_data):
        """
        Make predictions on new data.
        Args:
            adjusted_data: Input features DataFrame.
        Returns:
            Array of predicted values in original scale.
        """
        X_scaled = self.scaler_X.transform(adjusted_data)
        X_seq, _ = self.prepare_sequences(X_scaled, np.zeros(len(X_scaled)))
        predictions = self._create_predict_function(X_seq)
        return self.scaler_y.inverse_transform(predictions)

# Main analysis function.
def run_analysis(adjusted_data):
    print("Training the model...")
    predictor = EconomicHealthPredictor(lookback=12)
    history = predictor.fit(adjusted_data, epochs=20, validation_split=0.2, ver
bose=0)

    # Cross-validation.
    print("Performing cross-validation...")
    cv_metrics, cv_history = predictor.cross_validate(adjusted_data, epochs=20,
k=5)
    print("Cross-Validation Metrics: " + ", ".join(f"{metric}: {value:.3f}" for
metric, value in cv_metrics.items()))

    # Plot training history after cross-validation.
    plot_training_history(cv_history)

    # Making predictions.
    predictions = predictor.predict(adjusted_data)
    health_score = predictor.create_composite_health_score(adjusted_data)

    # Add model evaluation.
    metrics = model_evaluation(health_score[12:], predictions)

    # Print metrics.
    print("\nModel Eval. Metrics:", end=' ')
    print(", ".join(f"{metric}: {value:.3f}" for metric, value in metrics.items
()))
    print("\n")

    # Visualization of predictions.
    plot_predictions(health_score[12:], predictions, adjusted_data.index[12:])
    return predictor, history, predictions, health_score, metrics

# Performing the analysis.
if __name__ == "__main__":
    # Loading the data.
    adjusted_data = pd.read_csv("data/adjusted_data.csv", index_col=0, parse_da
tes=True)
    # Training and analysis.
```

```
    predictor, history, predictions, health_score, metrics = run_analysis(adjus
ted_data)

Training the model...
Performing cross-validation...
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 92ms/step
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 91ms/step
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 93ms/step
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 95ms/step
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 91ms/step
Cross-Validation Metrics: rmse: 0.063, mae: 0.048, r2: 0.938
```



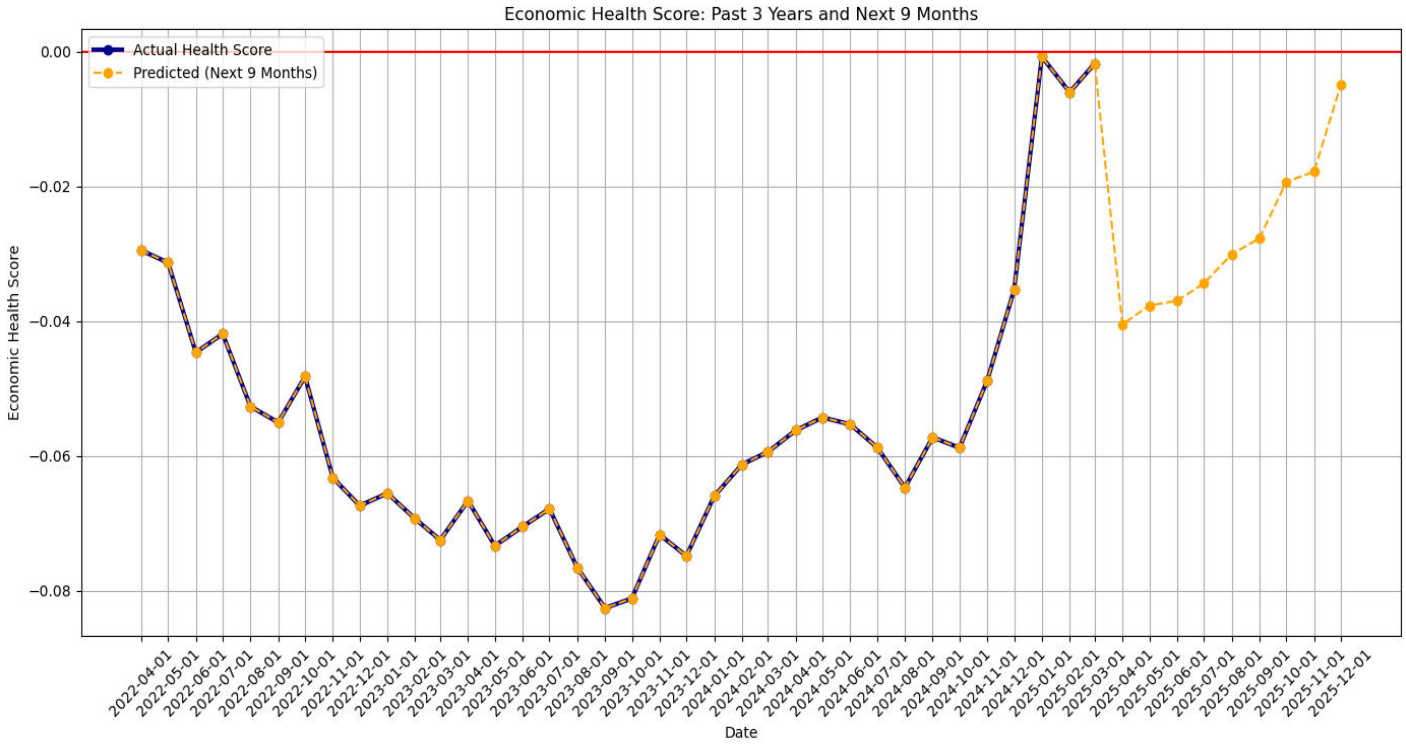

```
Model Eval. Metrics: rmse: 0.016, mae: 0.012, r2: 0.942
```



BINGO! Those who know the phases of development of the US economy **can now clearly determine the periods of rise, fall and stagnation! And they coincide with what historical truth says**. Let's try to predict what will happen in the next 9 months. Why 9? Because the data collection is delayed by 3 months. That is, we have 6 months of current forecast left.

```
def predict_next_months(predictor, adjusted_data, months=9):
    """
    Predict the next 'months' months using the trained predictor model.
    """
    # Extract the latest data for prediction.
    last_data = adjusted_data.iloc[-predictor.lookback:].values
    last_data_scaled = predictor.scaler_X.transform(last_data)
    predictions = []

    for _ in range(months):
        # Prepare the sequence.
        last_seq = last_data_scaled[-predictor.lookback:].reshape(1, predictor.
lookback, -1)

        # Predict the next month.
        next_prediction_scaled = predictor.model.predict(last_seq, verbose=0)[0
]
        next_prediction = predictor.scaler_y.inverse_transform(next_prediction_
scaled.reshape(-1, 1))[0, 0]
        predictions.append(next_prediction)

        # Update the sequence with the predicted value.
        next_data_scaled = np.append(last_data_scaled[-1, 1:], next_prediction_
scaled).reshape(1, -1)
        last_data_scaled = np.vstack((last_data_scaled, next_data_scaled))

    return predictions

# Forecast for next 9 months.
next_9_months_predictions = predict_next_months(predictor, adjusted_data, month
s=9)
# Create future dates (without gap between August and September).
future_dates = pd.date_range(start=adjusted_data.index[-1] + pd.offsets.MonthBe
gin(1), periods=9, freq='MS')
# Historical data (without last month to avoid gap).
historical_dates = adjusted_data.index[-36:]

# Forecast data should start from the next month after the last hist. month.
all_dates = historical_dates.append(future_dates)
# Combining historical values with forecasted ones.
extended_health_score = np.concatenate((health_score.values[-36:], next_9_month
s_predictions))

# Display the plot.
plt.figure(figsize=(14, 7))
# Historical data with darker blue color and thicker line.
plt.plot(
    historical_dates,
    health_score[-36:],
    label="Actual Health Score",
    color="darkblue",  # Darker blue color.
    linestyle="-",
    marker="o",
    linewidth=3  # Increased line thickness.
```

```
)
# Forecasted data.
plt.plot(
    all_dates,  # Combine historical and forecast dates.
    extended_health_score,  # Combine historical and forecast values.
    label="Predicted (Next 9 Months)",
    color="orange",
    linestyle="--",
    marker="o"
)
# Add zero line for better visualization.
plt.axhline(y=0, color='r', linestyle='-')
# Title and legend.
plt.title("Economic Health Score: Past 3 Years and Next 9 Months")
plt.xlabel("Date")
plt.ylabel("Economic Health Score")
plt.legend(loc='upper left')
# Change frequency of monthly labels on X axis.
plt.xticks(all_dates, rotation=45)
# Add grid for better visualization.
plt.grid(True)
# Improve visualization.
plt.tight_layout()
plt.show()
```



What can we read from the graph? The US economy will fluctuate around our virtual zero in the near future. Which means that it is not threatened by anything unusual. What's the last question we can ask ourselves? So far in this 3-part study, we've seen that the **economy is in good shape because of constant infusions of money from loans**. How long can this all continue? **How long** can the effect of "helicopter money" (US Debt Clock) last? According to economists dealing with the subject, **until the debts reach 200% of the Gross Domestic Product** (GDP). In a bad geopolitical situation up to 175%. [21] Of course, the geopolitical situation is very bad now, so we **are stopping at a value of 175%**.

4. Debt forecasting

4.1 Using Deep Learning model.

Input: GDP, W006RC1Q027SBEA, FEDFUNDS, CPIAUCSL, GFDEBTN, BOPGSTB, FGEXPND **Target**: Total Public Debt as Percent of Gross Domestic Product (GFDEGDQ188S) **Model**: The first one that was described. But only the features that are directly related to the Debt/GDP ratio were taken! **Objective**: To predict the future Debt/GDP ratio for 36 months ahead.

Positive:

- GDP (core economic growth indicator)
- W006RC1Q027SBEA (tax revenues)

Negative:

- GFDEBTN (total public debt)
- CPIAUCSL (inflation)
- FEDFUNDS (cost of borrowing)
- BOPGSTB (trade deficit)
- FGEXPND (government spending)

```
# Function to visualize model predictions against actual values.
def plot_predictions(true_values, predictions, dates):
    """
    Create a comparison plot of predicted vs actual values.
    Args:
        true_values: Array of actual values.
        predictions: Array of model predictions.
        dates: Array of corresponding dates for x-axis.
    """
    plt.figure(figsize=(15, 6))
    plt.plot(dates, true_values, label='Actual', alpha=0.7)
    plt.plot(dates, predictions, label='Predicted', alpha=0.7)
    plt.title('Predicted vs Actual Debt-to-GDP Ratio')
    plt.legend(loc='upper left')
    plt.tight_layout()
    plt.grid(True)
    plt.show()

# Economic health prediction class.
class EconomicHealthPredictor:
    def __init__(self, lookback=12):
        """
        Initialize predictor with specified lookback period.
        Args:
            lookback: Number of previous time steps to consider for prediction.
        """
        self.lookback = lookback
        self.scaler_X = MinMaxScaler()
        self.scaler_y = MinMaxScaler()
        self.model = None

    def prepare_sequences(self, X, y):
        """
```

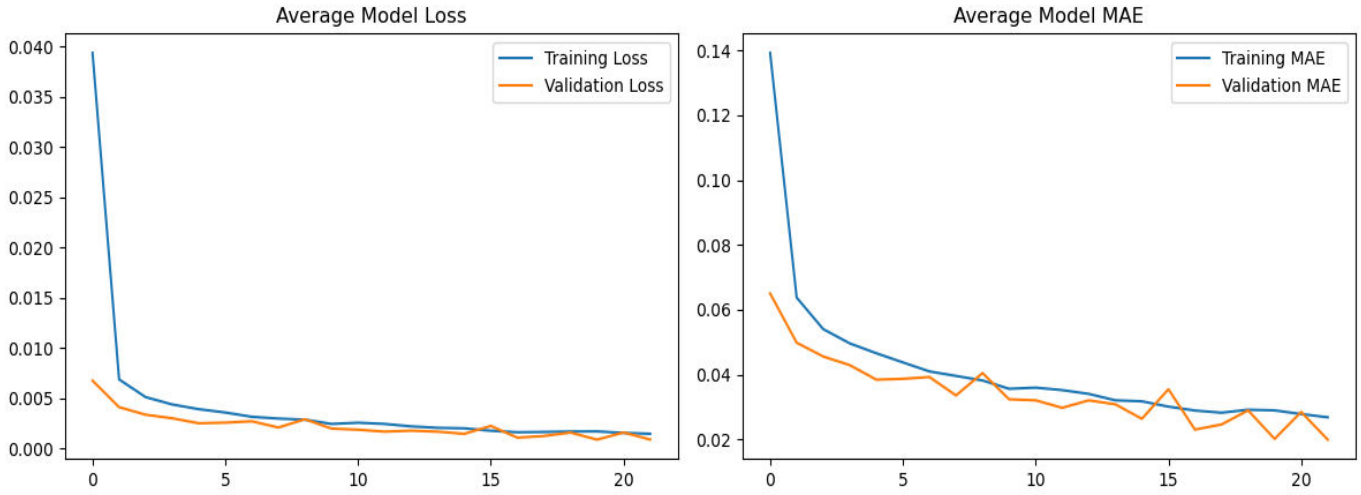
```
        Create sequences of lookback periods for time series prediction.
        Args:
            X: Input features array.
            y: Target values array.
        Returns:
            Tuple of (X sequences, y sequences) for model training.
        """
        X_seq, y_seq = [], []
        for i in range(len(X) - self.lookback):
            X_seq.append(X[i:(i + self.lookback)])
            y_seq.append(y[i + self.lookback])
        return np.array(X_seq), np.array(y_seq)

    def build_model(self, input_shape):
        """
        Creates a model.
        """
        model = Sequential([
            Conv1D(128, kernel_size=5, activation='relu', input_shape=input_sha
pe),
            MaxPooling1D(pool_size=2),
            LSTM(256, return_sequences=True),
            Dropout(0.3),
            LSTM(128, return_sequences=False),
            Dropout(0.3),
            Dense(64, activation='relu'),
            Dense(1)
        ])
        model.compile(optimizer='adam', loss='mse', metrics=['mae'])
        return model

    def fit(self, data, target_column, epochs=22, validation_split=0.2, verbose
=0):
        """
        Train the model on input data.
        Args:
            data: Input DataFrame with features and target.
            target_column: Name of the target column.
            epochs: Number of training epochs.
            validation_split: Fraction of data to use for validation.
            verbose: Verbosity level of training output.
        Returns:
            Training history.
        """
        # Select specified features and target.
        features = ['gdp', 'tax', 'fedfunds', 'cpi', 'debt', 'tb', 'fgexpnd']
        X = data[features]
        y = data[target_column]

        # Scale data.
        X_scaled = self.scaler_X.fit_transform(X)
        y_scaled = self.scaler_y.fit_transform(y.values.reshape(-1, 1))
```

```
        # Prepare sequences for time series prediction.
        X_seq, y_seq = self.prepare_sequences(X_scaled, y_scaled)
        self.model = self.build_model((self.lookback, X.shape[1]))

        # Early stopping to prevent overfitting.
        early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore
_best_weights=True)

        history = self.model.fit(
            X_seq, y_seq,
            epochs=epochs,
            validation_split=validation_split,
            callbacks=[early_stopping],
            verbose=verbose
        )
        return history

    def predict(self, data):
        """
        Make predictions on new data.
        Args:
            data: Input DataFrame with features.
        Returns:
            Array of predicted values in original scale.
        """
        features = ['gdp', 'tax', 'fedfunds', 'cpi', 'debt', 'tb', 'fgexpnd']
        X = data[features]
        X_scaled = self.scaler_X.transform(X)
        X_seq, _ = self.prepare_sequences(X_scaled, np.zeros(len(X_scaled)))
        predictions = self.model.predict(X_seq, verbose=0)
        return self.scaler_y.inverse_transform(predictions)

# Main analysis function.
def run_analysis(data):
    print("Running K-fold Cross-Validation...")
    target = 'debt_per_gdp'
    predictor = EconomicHealthPredictor(lookback=12)

    features = ['gdp', 'tax', 'fedfunds', 'cpi', 'debt', 'tb', 'fgexpnd']
    X = data[features]
    y = data[target]

    # Scale data.
    X_scaled = predictor.scaler_X.fit_transform(X)
    y_scaled = predictor.scaler_y.fit_transform(y.values.reshape(-1, 1))

    # Prepare sequences.
    X_seq, y_seq = predictor.prepare_sequences(X_scaled, y_scaled)

    # K-fold cross-validation.
    kf = KFold(n_splits=5, shuffle=True, random_state=42)
    fold_metrics = []
    histories = []
```

```
    for fold, (train_index, val_index) in enumerate(kf.split(X_seq), start=1):
        X_train, X_val = X_seq[train_index], X_seq[val_index]
        y_train, y_val = y_seq[train_index], y_seq[val_index]

        predictor.model = predictor.build_model((predictor.lookback, X.shape[1]
))

        history = predictor.model.fit(
            X_train, y_train,
            validation_data=(X_val, y_val),
            epochs=22,
            callbacks=[EarlyStopping(monitor='val_loss', patience=10, restore_b
est_weights=True)],
            verbose=0
        )

        # Store training history for later visualization.
        histories.append(history)

        # Evaluate on validation set.
        val_predictions = predictor.model.predict(X_val, verbose=0)
        metrics = model_evaluation(predictor.scaler_y.inverse_transform(y_val),
                                   predictor.scaler_y.inverse_transform(val_pre
dictions))
        fold_metrics.append(metrics)

    # Average metrics over folds.
    avg_metrics = {
        metric: np.mean([fold[metric] for fold in fold_metrics]) for metric in
fold_metrics[0]
    }

    print("\nAverage Cross-Validation Metrics:")
    for metric, value in avg_metrics.items():
        print(f"{metric}: {value:.3f}")

    # Plot average training history.
    avg_history = {
        'loss': np.mean([h.history['loss'] for h in histories], axis=0),
        'val_loss': np.mean([h.history['val_loss'] for h in histories], axis=0)
,
        'mae': np.mean([h.history['mae'] for h in histories], axis=0),
        'val_mae': np.mean([h.history['val_mae'] for h in histories], axis=0)
    }
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(avg_history['loss'], label='Training Loss')
    plt.plot(avg_history['val_loss'], label='Validation Loss')
    plt.title('Average Model Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(avg_history['mae'], label='Training MAE')
    plt.plot(avg_history['val_mae'], label='Validation MAE')
```

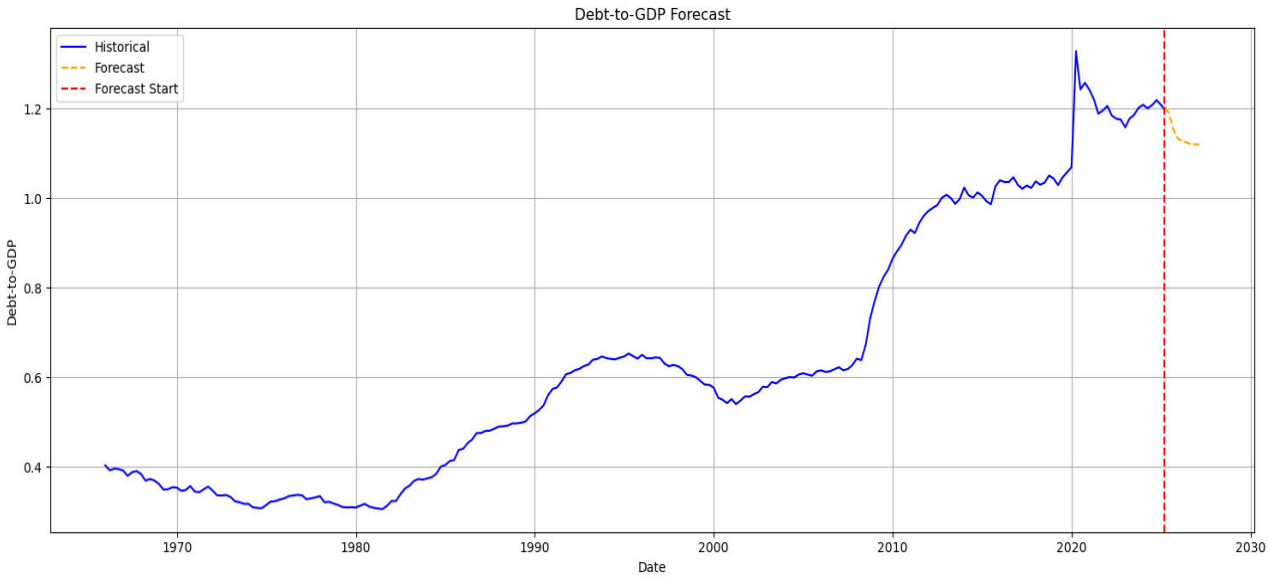
```
    plt.title('Average Model MAE')
    plt.legend()

    plt.tight_layout()
    plt.show()

    return predictor, avg_metrics

# Perform the analysis if executed as main script.
if __name__ == "__main__":
    # Example of loading the data.
    # data = pd.read_csv("data.csv", index_col=0, parse_dates=True)
    predictor, avg_metrics = run_analysis(data)
```

```
Running K-fold Cross-Validation...

Average Cross-Validation Metrics:
rmse: 0.026
mae: 0.018
r2: 0.992
```




```
def forecast_future(predictor, data, steps=60):
    """
    Forecast future values for the target variable.

    Args:
        predictor: Trained EconomicHealthPredictor object.
        data: DataFrame containing the historical data.
        steps: Number of months to forecast into the future.

    Returns:
        DataFrame with historical and forecasted values.
    """
    features = ['gdp', 'tax', 'fedfunds', 'cpi', 'debt', 'tb', 'fgexpnd']
    last_inputs = data[features].iloc[-predictor.lookback:].values
    last_inputs_scaled = predictor.scaler_X.transform(last_inputs)

    future_predictions = []
    for _ in range(steps):
        input_seq = last_inputs_scaled[-predictor.lookback:].reshape(1, predict
```

```
or.lookback, -1)
        next_pred_scaled = predictor.model.predict(input_seq, verbose=0)[0, 0]
        next_pred = predictor.scaler_y.inverse_transform([[next_pred_scaled]])[
0, 0]
        future_predictions.append(next_pred)

        # Update input sequence with the new prediction.
        new_input = last_inputs_scaled[-1].copy()
        new_input[-1] = next_pred_scaled  # Update 'debt_per_gdp' equivalent.
        last_inputs_scaled = np.vstack([last_inputs_scaled, new_input])

    future_dates = [data.index[-1] + timedelta(days=30 * (i + 1)) for i in rang
e(steps)]
    future_df = pd.DataFrame({'date': future_dates, 'forecast': future_predicti
ons})
    return future_df

# Forecast future values.
steps_to_forecast = 24
forecast_df = forecast_future(predictor, data, steps=steps_to_forecast)

# Combine historical and forecasted data.
historical_data = data[['debt_per_gdp']].reset_index()
historical_data.columns = ['date', 'actual']
combined_df = pd.concat([
    historical_data,
    forecast_df.rename(columns={'forecast': 'actual'})
])

# Visualize the forecast with a dotted orange line for predictions.
plt.figure(figsize=(15, 6))

# Plot historical actual values.
plt.plot(historical_data['date'], historical_data['actual'], label='Historical'
, color='blue')

# Plot forecasted values with dotted orange line.
plt.plot(forecast_df['date'], forecast_df['forecast'], linestyle='--', color='o
range', label='Forecast')

# Add a vertical line to separate historical and forecast data.
plt.axvline(x=pd.Timestamp('2025-03-01'), color='red', linestyle='--', label='F
orecast Start')

# Customize the plot.
plt.title('Debt-to-GDP Forecast')
plt.xlabel('Date')
plt.ylabel('Debt-to-GDP')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



4.2 Using econometrics.

Let's try to make a forecast for the debt of the US economy in a different way. Using the tools of econometrics. **Econometrics** uses **economic** theory, **mathematics**, and **statistical inference** to quantify economic phenomena. The model shown below uses a **vector autoregression (VAR) method** to analyze time series and make forecasts of economic indicators.

Purpose of the model:

- **To analyze the relationships** between gross domestic product (GDP), debt, and federal tax collected (GDP, debt, and tax).
- **To forecast their values** for a specified number of months in the future.

How it works:

- The model takes historical data for the above indicators.
- Learns how past values are related.
- Uses these relationships to predict future values.

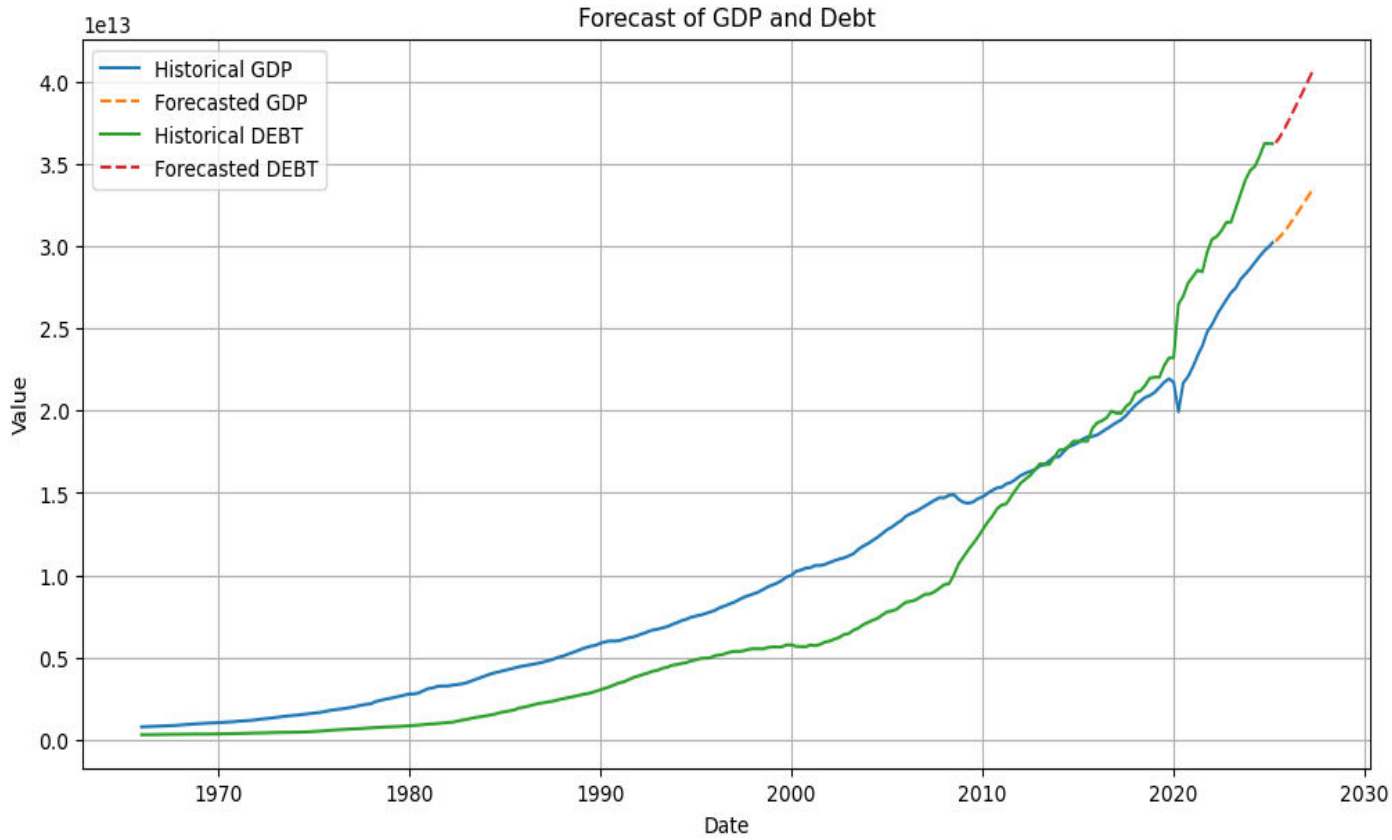
Output: Forecasts for the values of GDP, debt, and taxes collected. Forecasts for ratios such as:

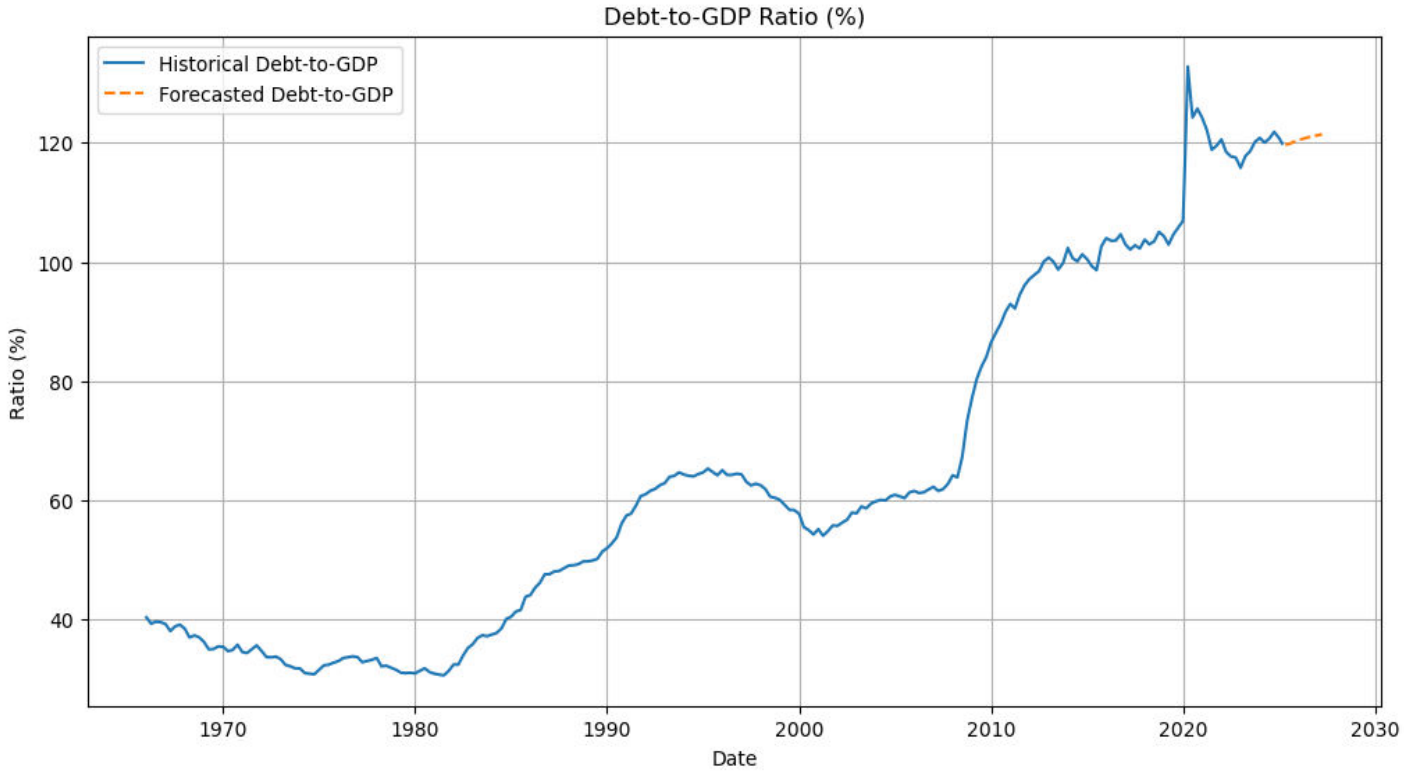
- Debt-to-GDP in percent.
- Debt-to-Tax in percent.

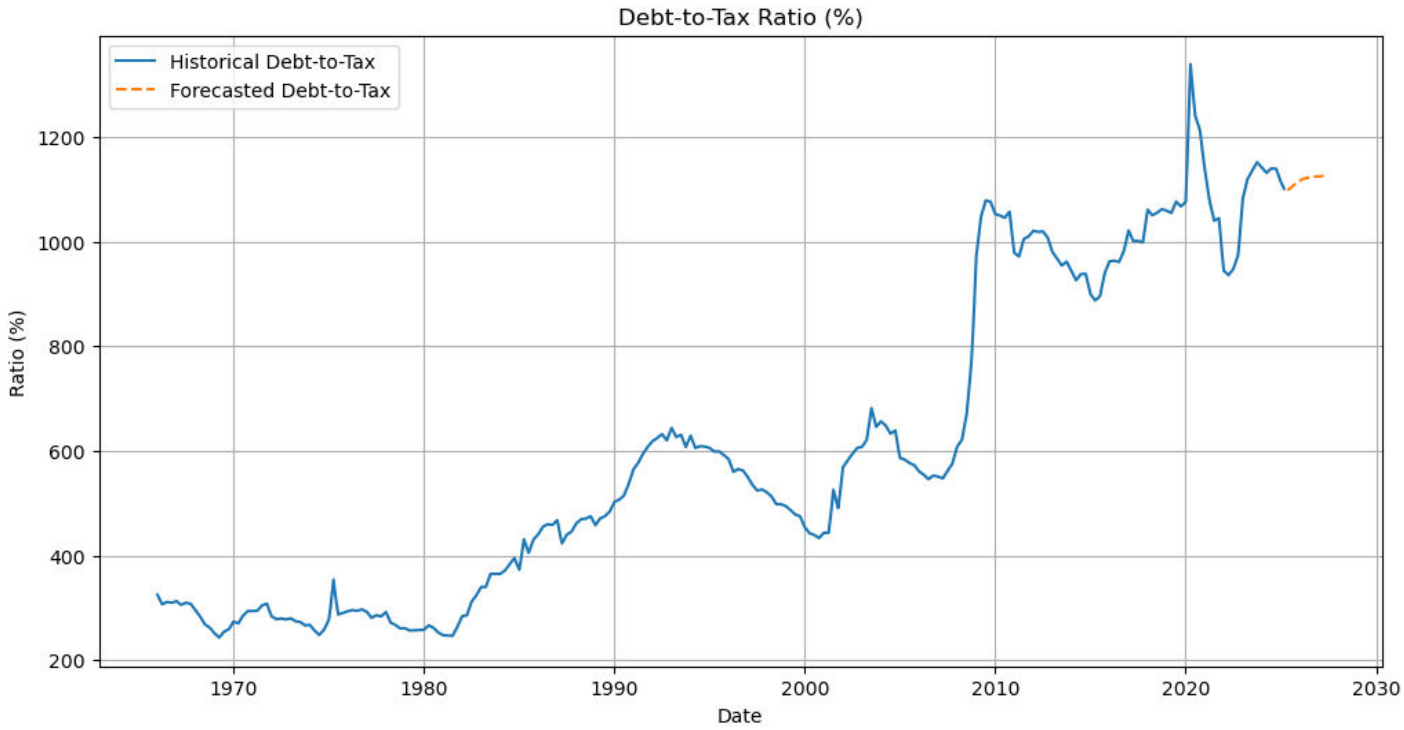
```
class VARModel:
    def __init__(self, lag_order=1):
        self.lag_order = lag_order
        self.model = None

    def fit(self, data):
        # Prepare the data for training.
        self.data = data
        self.model = VAR(self.data)
        self.results = self.model.fit(self.lag_order)
        # print(self.results.summary())

    def forecast(self, steps=24):
        # Generate forecasts for the specified number of steps.
        forecast = self.results.forecast(y=self.data.values[-self.lag_order:],
```

```
steps=steps)
        forecast_index = pd.date_range(start=self.data.index[-1], periods=steps
+1, freq='ME')[1:]
        forecast_df = pd.DataFrame(forecast, index=forecast_index, columns=self
.data.columns)

        # Calculate forecasted ratios.
        forecast_df['debt_to_gdp'] = (forecast_df['debt'] / forecast_df['gdp'])
* 100
        forecast_df['debt_to_tax'] = (forecast_df['debt'] / forecast_df['tax'])
* 100
        return forecast_df

    def plot_forecast(self, forecast_df):
        # Visualize forecasts for GDP and debt.
        plt.figure(figsize=(12, 6))

        for column in ['gdp', 'debt']:
            plt.plot(self.data.index, self.data[column], label=f'Historical {co
lumn.upper()}')
            plt.plot(forecast_df.index, forecast_df[column], linestyle='--', la
bel=f'Forecasted {column.upper()}')

        plt.title('Forecast of GDP and Debt')
        plt.xlabel('Date')
        plt.ylabel('Value')
        plt.legend()
        plt.grid(True)
        plt.show()

        # Visualize the debt-to-GDP ratio.
        plt.figure(figsize=(12, 6))

        debt_to_gdp = (self.data['debt'] / self.data['gdp']) * 100
        plt.plot(self.data.index, debt_to_gdp, label='Historical Debt-to-GDP')
        plt.plot(forecast_df.index, forecast_df['debt_to_gdp'], linestyle='--',
label='Forecasted Debt-to-GDP')

        plt.title('Debt-to-GDP Ratio (%)')
        plt.xlabel('Date')
        plt.ylabel('Ratio (%)')
        plt.legend()
        plt.grid(True)
        plt.show()

        # Visualize the debt-to-tax ratio.
        plt.figure(figsize=(12, 6))

        debt_to_tax = (self.data['debt'] / self.data['tax']) * 100
        plt.plot(self.data.index, debt_to_tax, label='Historical Debt-to-Tax')
        plt.plot(forecast_df.index, forecast_df['debt_to_tax'], linestyle='--',
label='Forecasted Debt-to-Tax')

        plt.title('Debt-to-Tax Ratio (%)')
```

```
        plt.xlabel('Date')
        plt.ylabel('Ratio (%)')
        plt.legend()
        plt.grid(True)
        plt.show()

        # Display the final forecasted values.
        print("\nFinal forecasted values:")
        print(forecast_df[['gdp', 'debt', 'debt_to_gdp', 'debt_to_tax']].iloc[-
1])

# Load the data.
data = pd.read_csv("data/data.csv", index_col=0, parse_dates=True)

# Set the frequency of the time index.
data.index = data.index.to_period('M').to_timestamp()

# Select relevant features.
selected_features = ['gdp', 'debt', 'tax']
data = data[selected_features]

# Train the VAR model.
var_model = VARModel(lag_order=2)
var_model.fit(data)

# Forecast the next 24 months.
forecast_df = var_model.forecast(steps=24)

# Visualize the forecasts.
var_model.plot_forecast(forecast_df)
```








```
Final forecasted values:
gdp            3.340271e+13
debt           4.056244e+13
debt_to_gdp    1.214346e+02
debt_to_tax    1.125638e+03
Name: 2027-03-31 00:00:00, dtype: float64
```

4.3 Using mathematical methods.

The econometric model is also not effective for forecasting over long time intervals. Let's use to simpler mathematical methods to understand under what situation the debt/GDP ratio will reach 175%. We use the following formula:

$$DEBT\_to\_GDP = DEBT\_to\_GDP \times \frac{1 + (FED\_RATE / 100)}{1 + (GDP\_GROWTH\_RATE / 100)}$$

For our scenario, we **cannot use interest** rates **greater than 6%**. According to the law of large numbers, with such indebtedness this will quickly lead to bankruptcy. For the same reason, we **cannot assume economic growth greater than 3%**. In the first part of this analysis, this problem was analyzed in more detail. It is simply that interest payments do not imply such luxury.

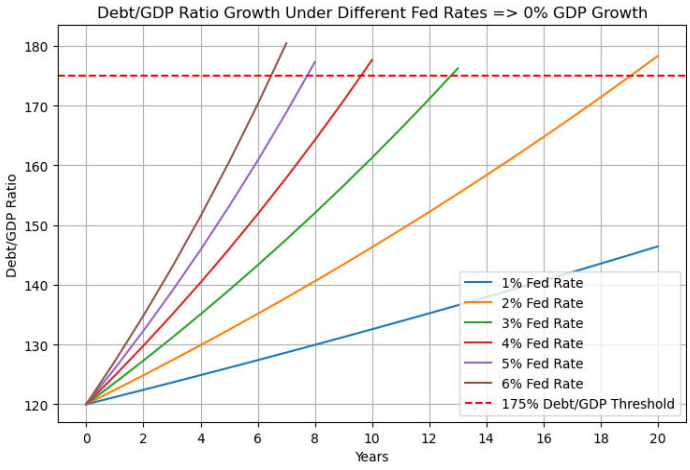
```
# Load the data.
data = pd.read_csv("data/data.csv")

# Define target debt/GDP ratio.
target_threshold = 175  # 175%

# Define current debt/GDP ratio (as of September 2024).
current_debt_to_gdp = 120  # 120%

# Define different Fed rates to test.
fed_rates = [1, 2, 3, 4, 5, 6]  # Fed rates in percentages.

# Define different annual GDP growth rates.
gdp_growth_rates = [0, 1, 2, 3]  # GDP growth rates in percentages.

# Initialize results.
results = {}

# Loop through each Fed rate.
for fed_rate in fed_rates:
    for gdp_growth_rate in gdp_growth_rates:
        # Initialize variables.
        debt_to_gdp = current_debt_to_gdp  # Start from current debt/GDP ratio.
        years = 0  # Number of years.

        # Simulate debt/GDP ratio growth.
        while debt_to_gdp < target_threshold and years < 20:  # Limit to 20 y.
            # Update debt/GDP ratio considering Fed rate and GDP growth.
            debt_to_gdp = debt_to_gdp * (1 + (fed_rate / 100)) / (1 + (gdp_grow
th_rate / 100))
            years += 1

        # Store results.
        key = (fed_rate, gdp_growth_rate)
        results[key] = {
            "Years until 175% debt/GDP": years,
            "Final predicted debt/GDP ratio": debt_to_gdp
        }

# Print results.
for (rate, growth), result in results.items():
```

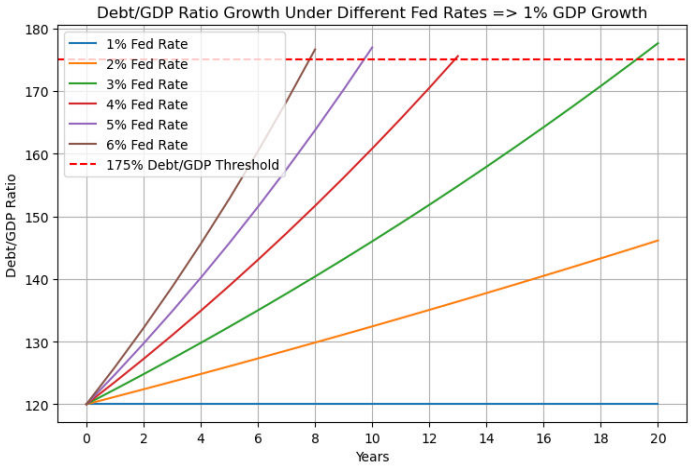
```
    print(f"At {rate}% Fed Rate and {growth}% GDP Growth:")
    if result['Final predicted debt/GDP ratio'] >= 175:
        print(f"Years until 175% debt/GDP: {result['Years until 175% debt/GDP']
}")
    print(f"Final predicted debt/GDP ratio: {result['Final predicted debt/GDP r
atio']:.2f}")
    print("-" * 42)

# Data visualization.
plt.figure(figsize=(15, 10))

for i, gdp_growth_rate in enumerate(gdp_growth_rates):
    plt.subplot(2, 2, i + 1)
    for fed_rate in fed_rates:
        debt_to_gdp = current_debt_to_gdp  # Start from current debt/GDP ratio.
        years_list = [0]  # List of years starting from 0.
        debt_to_gdp_list = [debt_to_gdp]  # List with initial debt/GDP ratio.
        years = 0

        while debt_to_gdp < target_threshold and years < 20:
            debt_to_gdp = debt_to_gdp * (1 + (fed_rate / 100)) / (1 + (gdp_grow
th_rate / 100))
            years_list.append(years + 1)  # Add next year.
            debt_to_gdp_list.append(debt_to_gdp)  # Add new debt/GDP ratio.
            years += 1

        plt.plot(years_list, debt_to_gdp_list, label=f"{fed_rate}% Fed Rate")

    plt.axhline(y=target_threshold, color="red", linestyle="--", label="175% De
bt/GDP Threshold")
    plt.xlabel("Years")
    plt.ylabel("Debt/GDP Ratio")
    plt.title(f"Debt/GDP Ratio Growth Under Different Fed Rates => {gdp_growth_
rate}% GDP Growth")
    plt.legend()
    plt.xticks(range(0, 21, 2))  # Show years from 0 to 20 in steps of 2.
    plt.grid()

plt.tight_layout()
plt.show()
```

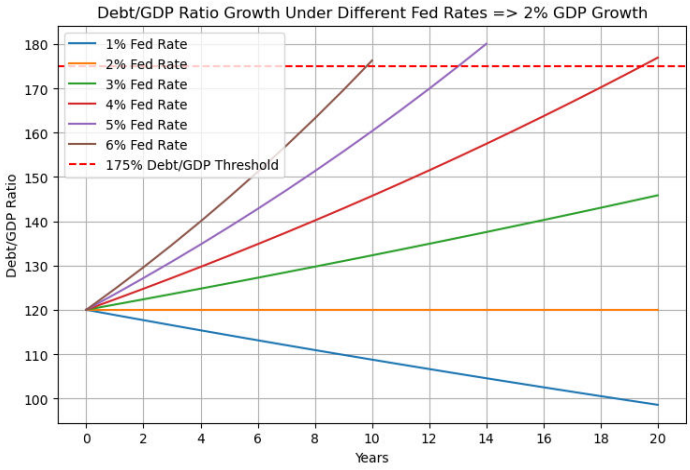
```
At 1% Fed Rate and 0% GDP Growth:
Final predicted debt/GDP ratio: 146.42
------------------------------------------
At 1% Fed Rate and 1% GDP Growth:
Final predicted debt/GDP ratio: 120.00
------------------------------------------
At 1% Fed Rate and 2% GDP Growth:
Final predicted debt/GDP ratio: 98.54
------------------------------------------
At 1% Fed Rate and 3% GDP Growth:
Final predicted debt/GDP ratio: 81.07
------------------------------------------
At 2% Fed Rate and 0% GDP Growth:
```

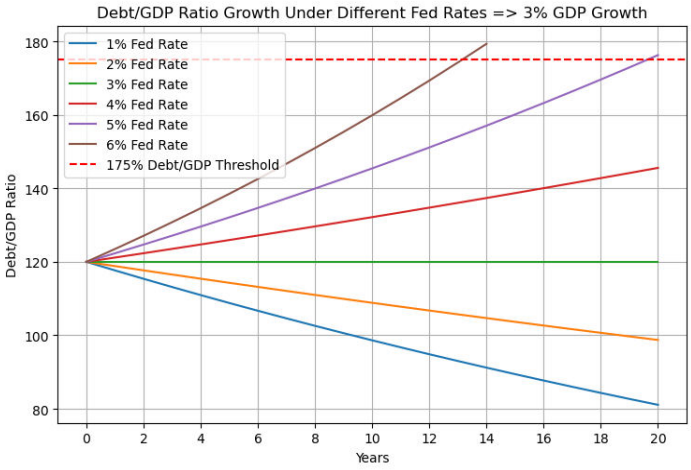
```
Years until 175% debt/GDP: 20
Final predicted debt/GDP ratio: 178.31
----------------------------------------
At 2% Fed Rate and 1% GDP Growth:
Final predicted debt/GDP ratio: 146.14
----------------------------------------
At 2% Fed Rate and 2% GDP Growth:
Final predicted debt/GDP ratio: 120.00
----------------------------------------
At 2% Fed Rate and 3% GDP Growth:
Final predicted debt/GDP ratio: 98.73
----------------------------------------
At 3% Fed Rate and 0% GDP Growth:
Years until 175% debt/GDP: 13
Final predicted debt/GDP ratio: 176.22
----------------------------------------
At 3% Fed Rate and 1% GDP Growth:
Years until 175% debt/GDP: 20
Final predicted debt/GDP ratio: 177.62
----------------------------------------
At 3% Fed Rate and 2% GDP Growth:
Final predicted debt/GDP ratio: 145.86
----------------------------------------
At 3% Fed Rate and 3% GDP Growth:
Final predicted debt/GDP ratio: 120.00
----------------------------------------
At 4% Fed Rate and 0% GDP Growth:
Years until 175% debt/GDP: 10
Final predicted debt/GDP ratio: 177.63
----------------------------------------
At 4% Fed Rate and 1% GDP Growth:
Years until 175% debt/GDP: 13
Final predicted debt/GDP ratio: 175.56
----------------------------------------
At 4% Fed Rate and 2% GDP Growth:
Years until 175% debt/GDP: 20
Final predicted debt/GDP ratio: 176.95
----------------------------------------
At 4% Fed Rate and 3% GDP Growth:
Final predicted debt/GDP ratio: 145.58
----------------------------------------
At 5% Fed Rate and 0% GDP Growth:
Years until 175% debt/GDP: 8
Final predicted debt/GDP ratio: 177.29
----------------------------------------
At 5% Fed Rate and 1% GDP Growth:
Years until 175% debt/GDP: 10
Final predicted debt/GDP ratio: 176.95
----------------------------------------
At 5% Fed Rate and 2% GDP Growth:
Years until 175% debt/GDP: 14
Final predicted debt/GDP ratio: 180.06
----------------------------------------
```

```
At 5% Fed Rate and 3% GDP Growth:
Years until 175% debt/GDP: 20
Final predicted debt/GDP ratio: 176.29
----------------------------------------
At 6% Fed Rate and 0% GDP Growth:
Years until 175% debt/GDP: 7
Final predicted debt/GDP ratio: 180.44
----------------------------------------
At 6% Fed Rate and 1% GDP Growth:
Years until 175% debt/GDP: 8
Final predicted debt/GDP ratio: 176.63
----------------------------------------
At 6% Fed Rate and 2% GDP Growth:
Years until 175% debt/GDP: 10
Final predicted debt/GDP ratio: 176.29
----------------------------------------
At 6% Fed Rate and 3% GDP Growth:
Years until 175% debt/GDP: 14
Final predicted debt/GDP ratio: 179.37
----------------------------------------
```









5. Conclusions

1. In this study, we compiled an economic health indicator from multiple indicators provided by the U.S. economy. And only **after we ran the features measured in dollars through a deflator did the true state of the U.S. economy become apparent** - the past, the present, and a little bit of the future.

2. The **second** Deep Learning **model predicts** for the near future (9 months ahead). The state of the **economy will be at almost the same level** as the current one.
3. The **latest** Deep Learning **model predicts** the future **Debt/GDP ratio**, but for 2 years ahead. The model does not predict any particular change other than the current state.
4. Using **econometrics**, we saw that the US economy and debt will continue to grow, but their **ratio will change slightly** in the direction of greater indebtedness. From the graph of the **Debt-to-Tax ratio**, we see that the debt exceeds taxes collected by over **11 times** (1125%). Now and in the near future. Such debt is extremely difficult to repay, even in some distant future. We do not even mention the interest on it.
5. The last 4 graphs show the following:
    - This is of course, **if interest rates and GDP growth follow some stable average movement** around the indicated percentages.
    - If the economy had **0% growth**, the debt-to-GDP ratio would reach 175% at a **6% interest** rate in just **over 6 years**, and at a **2% interest** rate in **about 19 years**.
    - If the economy has **1% growth**, the debt/GDP ratio would reach 175% at **6% interest** in **about 8 years**, and at **3% interest** in **about 19 years**.
    - f the economy has **2% growth**, the debt/GDP ratio would reach 175% at **6% interest** in **about 10 years**, and at **4% interest** in **over 19 years**.
    - If the economy has **3% growth**, the debt/GDP ratio would reach 175% at **6% interest** in **about 13 years**, and at **5% interest** in **about 20 years**.
6. Therefore, for the **US economy to escape bankruptcy over a 20-year period, it must maintain low interest rates (up to 4%) and at least 2% economic growth**. Because the geopolitical situation is unlikely to improve during this period.
7. But from the first and second parts of this study, we saw that US industrial production has currently reached its ceiling. The burden of interest rates is becoming greater and in order to compensate for this (i.e., to have economic growth), the service economy must grow at a very high rate. How this happens, given that the real economy is not growing - I do not know. However, US President Donald Trump knows: first trade wars, then we will see...
8. Some of the data was obtained from a future approximation. The exact historical data will be out in about 3 months. If we want to be perfect - we have to run the models with the latest data. Thanks for your attention :)

Resources:

18. **Sustainable Economic Indicators**: https://www.sustainable-environment.org.uk/Indicators/Economy.php
19. **GROSS DOMESTIC PRODUCT (GDP)**: https://fred.stlouisfed.org/series/GDP
20. **Consumer Price Index (CPI)**: https://fred.stlouisfed.org/series/CPIAUCSL
21. **Production Index (INDPRO)**: https://fred.stlouisfed.org/series/INDPRO
22. **S&P 500 Index**: https://stooq.com/q/d/?s=%5Espx&c=0&d1=19620101&d2=20241001&i=m
23. **Total Nonfarm Payroll (PAYEMS)**: https://fred.stlouisfed.org/series/PAYEMS
24. **UNEMPLOYMENT (UNRATE)**: https://fred.stlouisfed.org/series/UNRATE
25. **Federal Funds Effective Rate (FEDFUNDS)**: https://fred.stlouisfed.org/series/FEDFUNDS
26. **Total Public Debt (GFDEBTN)**: https://fred.stlouisfed.org/series/GFDEBTN
27. **Personal Saving Rate (PSAVERT)**: https://fred.stlouisfed.org/graph/?g=580A
28. **Real GDP per capita (A939RX0Q048SBEA)**: https://fred.stlouisfed.org/series/A939RX0Q048SBEA
29. **Total Public Debt as Percent of GDP (GFDEGDQ188S)**: https://fred.stlouisfed.org/series/GFDEGDQ188S

30. **Trade Balance (BOPGSTB)**: https://fred.stlouisfed.org/series/BOPGSTB
31. **United States Balance of Trade**: https://tradingeconomics.com/united-states/balance-of-trade
32. **Energy goods and services (DNRGRC1M027SBEA)**: https://fred.stlouisfed.org/series/DNRGRC1M027SBEA
33. **Interest payments (A091RC1Q027SBEA)**: https://fred.stlouisfed.org/series/A091RC1Q027SBEA
34. **Current tax receipts (W006RC1Q027SBEA)**: https://fred.stlouisfed.org/series/W006RC1Q027SBEA
35. **Slope of the yield curve (T10YFF)**: https://fred.stlouisfed.org/series/T10YFF
36. **Federal Government: Current Expenditures (FGEXPND)**: https://fred.stlouisfed.org/series/FGEXPND
37. **Purchasing Power of the Consumer Dollar in U.S. (CUUR0000SA0R)**: https://fred.stlouisfed.org/series/CUUR0000SA0R
38. **Bloomberg: US stuck in debt denial** https://opposition.bg/bloomberg-sasht-zamryaha-v-sastoyanie-na-otritsanie-na-dalga/

Calculations

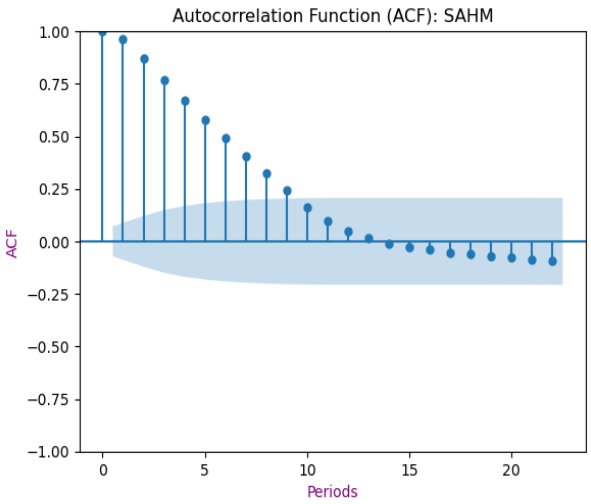
```
# import libraries
import pandas as pd
import numpy as np
import plotly.express as px
import matplotlib.pyplot as plt

from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.stattools import adfuller

collected_data = pd.read_csv('data/collected_data.csv') # Read the CSV file.

keep_col = ['sahm', 'indpro', 'sp500', 'tr10', 't10yff', 'unrate', 'pcepi', 'pa
yems', 'houst', 'recession']
all_data = collected_data[keep_col]
del collected_data

all_data
```

```
     sahm    indpro     sp500      tr10    t10yff  unrate  pcepi    payems  \
0   -0.17  0.016229  0.016139 -0.043737  1.650556   0.055  0.042  0.827913
1   -0.17  0.005350 -0.005878 -0.108990  1.078182   0.056  0.021  0.819544
2   -0.10  0.002130 -0.063973 -0.087455  1.043000   0.056  0.018  0.829109
3   -0.07 -0.001066 -0.089914  0.030636  1.441818   0.055  0.010  0.817113
4    0.00 -0.002132 -0.085381  0.035411  1.206667   0.055  0.010  0.816714
..    ...       ...       ...       ...       ...     ...    ...       ...
747  0.37  0.006954  0.046904 -0.056818 -0.847727   0.040 -0.010  0.824685
748  0.43  0.001500  0.034082 -0.177010 -1.024737   0.041  0.145  0.820780
749  0.53 -0.006214  0.011258 -0.056627 -1.081364   0.043  0.202  0.821816
750  0.57  0.003373  0.022578 -0.377727 -1.459091   0.042  0.142  0.822414
751  0.50 -0.002832  0.019996 -0.147409 -1.406500   0.041  0.217  0.826200

        houst  recession
0    0.396825          0
1    0.478671          0
2    0.518849          0
3    0.498512          0
4    0.459325          0
..        ...        ...
747  0.415179          0
748  0.422123          0
749  0.388889          0
750  0.437996          0
751  0.434524          0

[752 rows x 10 columns]
```

```
# We select the column we want to analyze.
time_series = all_data['sahm'].copy()
# Checking for stationarity with the ADF test.
```

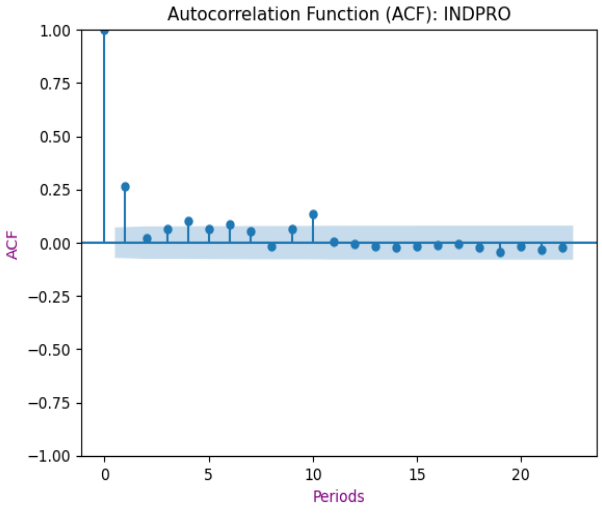
```
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stationary, we can make it stat. by differentiation
# time_series.diff().dropna()

# If the p-value is greater than 0.05, the series is not stationary.
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)

# We want to see the autocorrelation for up to 22 lag periods.
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22)

plt.title('Autocorrelation Function (ACF): SAHM')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -5.617273985585307
p-value: 1.1666263101298397e-06
Critical Value (1%): -3.439314999916068
Critical Value (5%): -2.8654965012008677
Critical Value (10%): -2.5688768817372867
Reject the null hypothesis (H0), the data does not have a unit root and is
stationary.
```



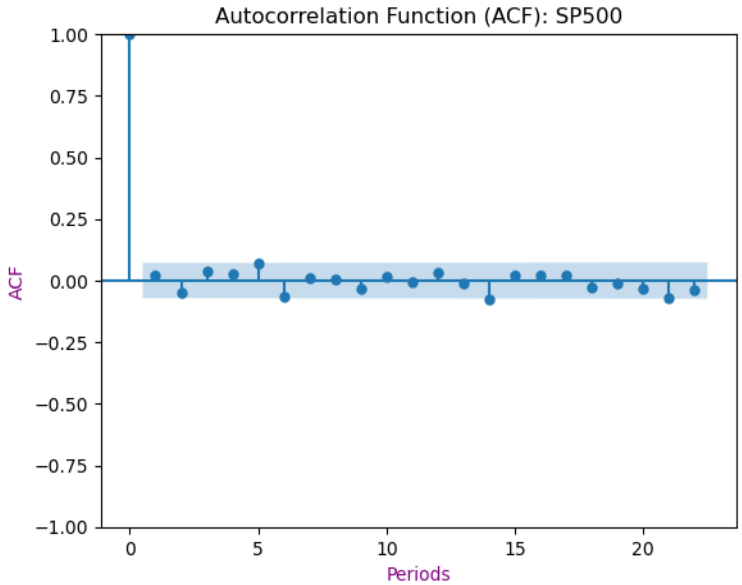
```
# We select the column we want to analyze.
time_series = all_data['indpro'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stat., we can make it stat. by differentiation!
# time_series.diff().dropna()
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorrelation for up to 2
2 lag periods.
plt.title('Autocorrelation Function (ACF): INDPRO')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -6.704029836066358
p-value: 3.825203796783599e-09
Critical Value (1%): -3.4392057325732104
Critical Value (5%): -2.8654483492874236
Critical Value (10%): -2.5688512291811225
Reject the null hypothesis (H0), the data does not have a unit root and is
stationary.
```



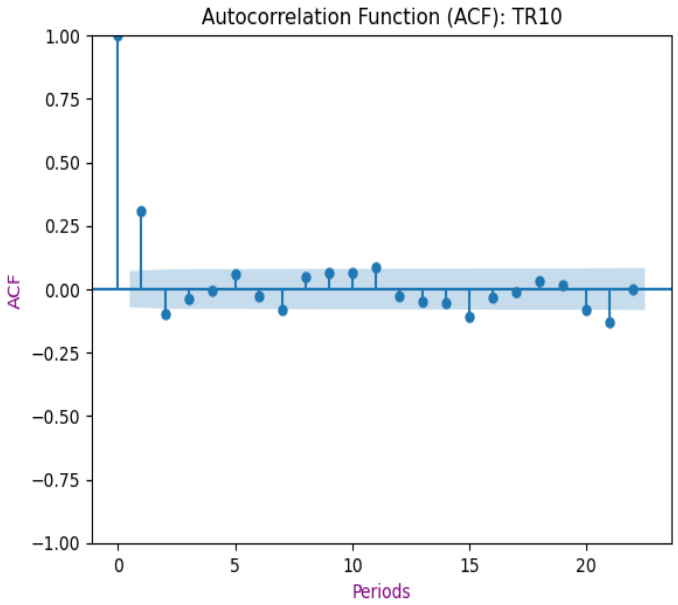
```
# We select the column we want to analyze.
time_series = all_data['sp500'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stat., we can make it stat. by differentiation!
# time_series.diff().dropna()
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorr. for up to 22 lags
plt.title('Autocorrelation Function (ACF): SP500')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -11.014868635450233
p-value: 6.193002403817569e-20
Critical Value (1%): -3.439146171679794
Critical Value (5%): -2.865422101274577
Critical Value (10%): -2.568837245865348
Reject the null hypothesis (H0), the data does not have a unit root and is
stationary.
```



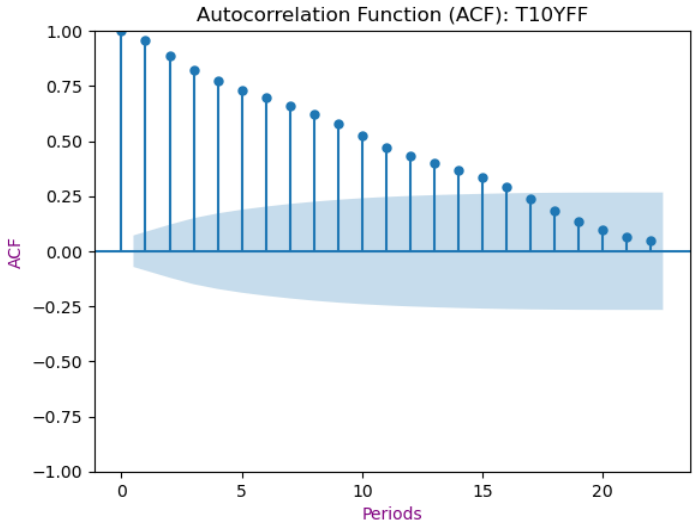
```
# We select the column we want to analyze.
time_series = all_data['tr10'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stat., we can make it stat. by differentiation!
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorr. for up to 22 lags
plt.title('Autocorrelation Function (ACF): TR10')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -7.321507860453668
p-value: 1.1908127058855779e-10
Critical Value (1%): -3.4393273074073045
Critical Value (5%): -2.8655019247555154
Critical Value (10%): -2.568879771109793
Reject the null hypothesis (H0), the data does not have a unit root and is
stationary.
```



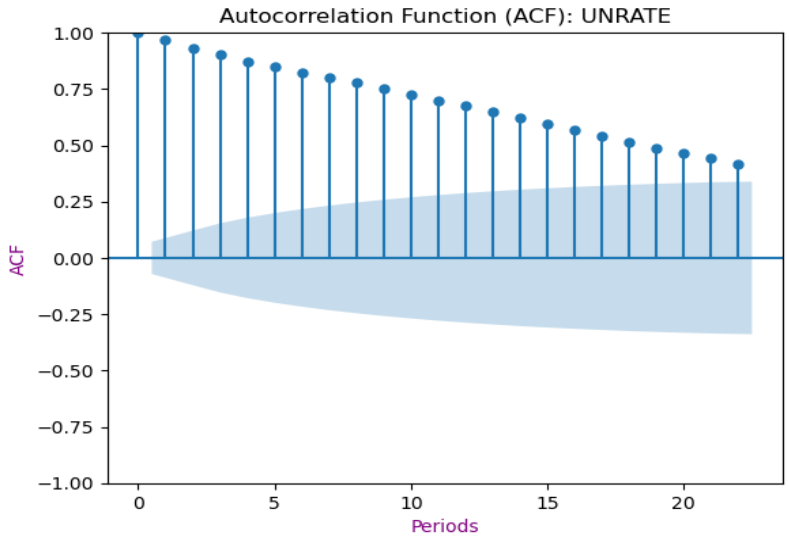
```
# We select the column we want to analyze.
time_series = all_data['t10yff'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stationary, we can make it stationary by differenti
ation!
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorrelation for up to 2
2 lag periods.
plt.title('Autocorrelation Function (ACF): T10YFF')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -5.2464075489585476
p-value: 7.063553789236375e-06
Critical Value (1%): -3.4392782790913206
Critical Value (5%): -2.865480319267325
Critical Value (10%): -2.568868260909806
Reject the null hypothesis (H0), the data does not have a unit root and is
stationary.
```



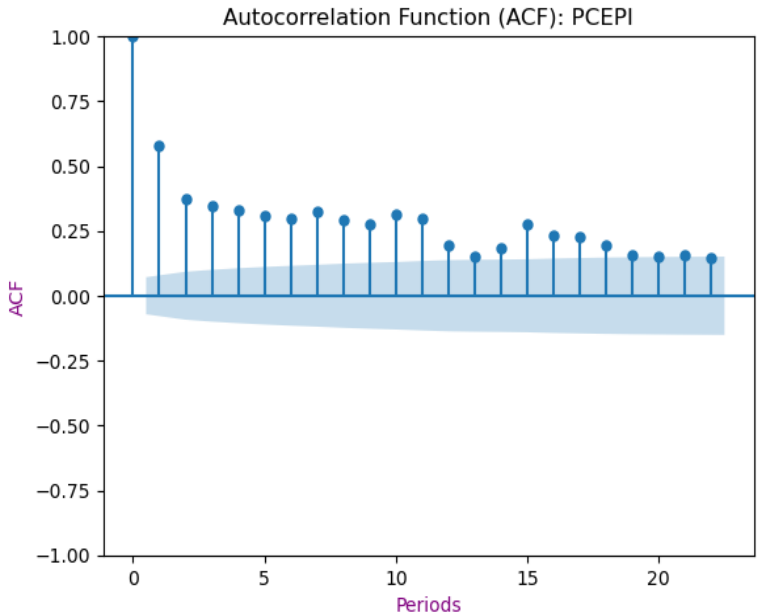
```
# We select the column we want to analyze.
time_series = all_data['unrate'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stat., we can make it stat. by differentiation!
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorr. for up to 22 lags
plt.title('Autocorrelation Function (ACF): UNRATE')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -3.2883393471462017
p-value: 0.015407820370500773
Critical Value (1%): -3.439110818166223
Critical Value (5%): -2.8654065210185795
Critical Value (10%): -2.568828945705979
Reject the null hypothesis (H0), the data does not have a unit root and is
stationary.
```



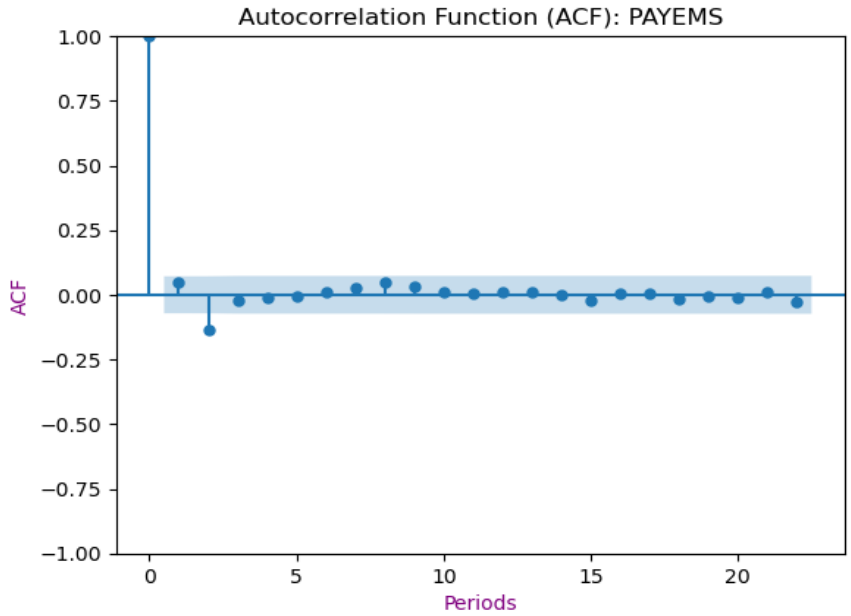
```
# We select the column we want to analyze.
time_series = all_data['pcepi'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stat., we can make it stat. by differentiation!
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorr. for up to 22 lags
plt.title('Autocorrelation Function (ACF): PCEPI')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -3.738859573536434
p-value: 0.0035985999579130792
Critical Value (1%): -3.4392539652094154
Critical Value (5%): -2.86546960465041
Critical Value (10%): -2.5688625527782327
Reject the null hypothesis (H0), the data does not have a unit root and is
stationary.
```



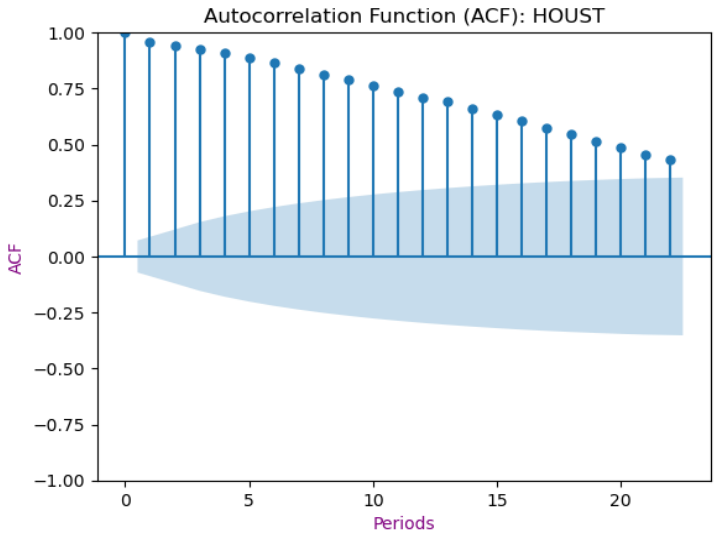
```
# We select the column we want to analyze.
time_series = all_data['payems'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stat., we can make it stat. by differentiation!
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorr. for up to 22 lags
plt.title('Autocorrelation Function (ACF): PAYEMS')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -21.67465711940653
p-value: 0.0
Critical Value (1%): -3.439099096730074
Critical Value (5%): -2.8654013553540745
Critical Value (10%): -2.568826193777778
Reject the null hypothesis (H0), the data does not have a unit root and is stat
ionary.
```



```
# We select the column we want to analyze.
time_series = all_data['houst'].copy()
# Checking for stationarity with the ADF test.
result = adfuller(time_series)
print('ADF Statistic:', result[0])
print('p-value:', result[1])
for key, value in result[4].items():
    print('Critical Value ({}): {}'.format(key, value))

# Differentiation of the time series:
# If the time series is not stat., we can make it stat. by differentiation!
if result[1] > 0.05:
    time_series = time_series.diff().dropna()

if result[1] > 0.05014:
    print('\n Fail to reject the null hypothesis (H0), the data has a unit root
and is non-stationary. \n')
else:
    print('\n Reject the null hypothesis (H0), the data does not have a unit ro
ot and is stationary. \n')

# Autocorrelation study (ACF)
plt.figure(figsize=(10, 6))
plot_acf(time_series, lags=22) # We want to see the autocorr. for up to 22 lags
plt.title('Autocorrelation Function (ACF): HOUST')
plt.xlabel("Periods", color = "purple")
plt.ylabel("ACF", color = "purple")
plt.show()
```

```
ADF Statistic: -3.876052680405001
p-value: 0.0022183279843288077
Critical Value (1%): -3.4392782790913206
Critical Value (5%): -2.865480319267325
Critical Value (10%): -2.568868260909806
Reject the null hypothesis (H0), the data does not have a unit root and is stat
ionary.
```



Modeling 1

```
# import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
# LogisticRegression
from sklearn.model_selection import StratifiedKFold, cross_val_predict, GridSea
rchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_s
core, confusion_matrix
from sklearn.preprocessing import StandardScaler
# Decision Tree
from sklearn.tree import DecisionTreeClassifier
# RandomForest
from sklearn.ensemble import RandomForestClassifier

collected_data = pd.read_csv('data/collected_data.csv', index_col=0) # CSV file
data = collected_data.copy()
# Setting column 'date' as index.
data = data.set_index('date', drop=True)
del collected_data

data
```

date	sahm	indpro	sp500	tr10	t10yff	unrate	pcepi	payems	houst	recession
1962-02-01	-0.17	0.016229	0.016139	-0.043737	1.650556	0.055	0.042	0.827913	0.396825	0
1962-03-01	-0.17	0.005350	-0.005878	-0.108990	1.078182	0.056	0.021	0.819544	0.478671	0
1962-04-01	-0.10	0.002130	-0.063973	-0.087455	1.043000	0.056	0.018	0.829109	0.518849	0
1962-05-01	-0.07	-0.001066	-0.089914	0.030636	1.441818	0.055	0.010	0.817113	0.498512	0
1962-06-01	0.00	-0.002132	-0.085381	0.035411	1.206667	0.055	0.010	0.816714	0.459325	0
...	...	...	...	...	...	...	...	...	...	...
2024-05-01	0.37	0.006954	0.046904	-0.056818	-0.847727	0.040	-0.010	0.824685	0.415179	0
2024-06-01	0.43	0.001500	0.034082	-0.177010	-1.024737	0.041	0.145	0.820780	0.422123	0
2024-07-01	0.53	-0.006214	0.011258	-0.056627	-1.081364	0.043	0.202	0.821816	0.388889	0
2024-08-01	0.57	0.003373	0.022578	-0.377727	-1.459091	0.042	0.142	0.822414	0.437996	0
2024-09-01	0.50	-0.002832	0.019996	-0.147409	-1.406500	0.041	0.217	0.826200	0.434524	0

```
[752 rows x 10 columns]
```

```
# get X and y
X = data.drop(['recession'], axis=1)
y = data['recession']

# We define the training period.
X_train, y_train = X.loc["1962-02-01":"2012-12-01"], y.loc["1962-02-01":"2012-12-01"]
# We define the test period.
# X_test, y_test = X.loc["2013-01-01":], y.loc["2013-01-01":]
```

1. Metrics

Here are the main metrics we can use to get the **final score** for each **model**:

1. **Precision**: It measures the proportion of true positive predictions relative to all positive predictions. It is **important** when you **want to minimize false positives**.Precision = TP/(TP+FP)
2. **Recall** (Sensitivity): It measures the proportion of true positive predictions to all actual positive cases. It is useful when you want to minimize false negatives.Recall = TP/(TP+FN)
3. **F1-score**: Is a metric that balances precision and recall. It is calculated as the harmonic mean of precision and recall. F1 Score is useful when seeking a balance between high precision and high recall, as it penalizes extreme negative values of either component.F1 = 2*Precision*Recall/(Precision+Recall)
4. **Confusion Matrix**: Visualizes true and predicted classes, which can help better understand model performance.
5. **Specificity**: It measures the proportion of true negative predictions relative to all actual negative cases. It is useful when you want to **minimize false positives**.Specificity = TN/(TN+FP)
6. **Accuracy**: It measures the proportion of correctly predicted cases (both positive and negative) relative to the total number of cases. Accuracy **can be misleading with unbalanced data**, as it can be high even if the model does not predict the small class well.

2. Logistic Regression

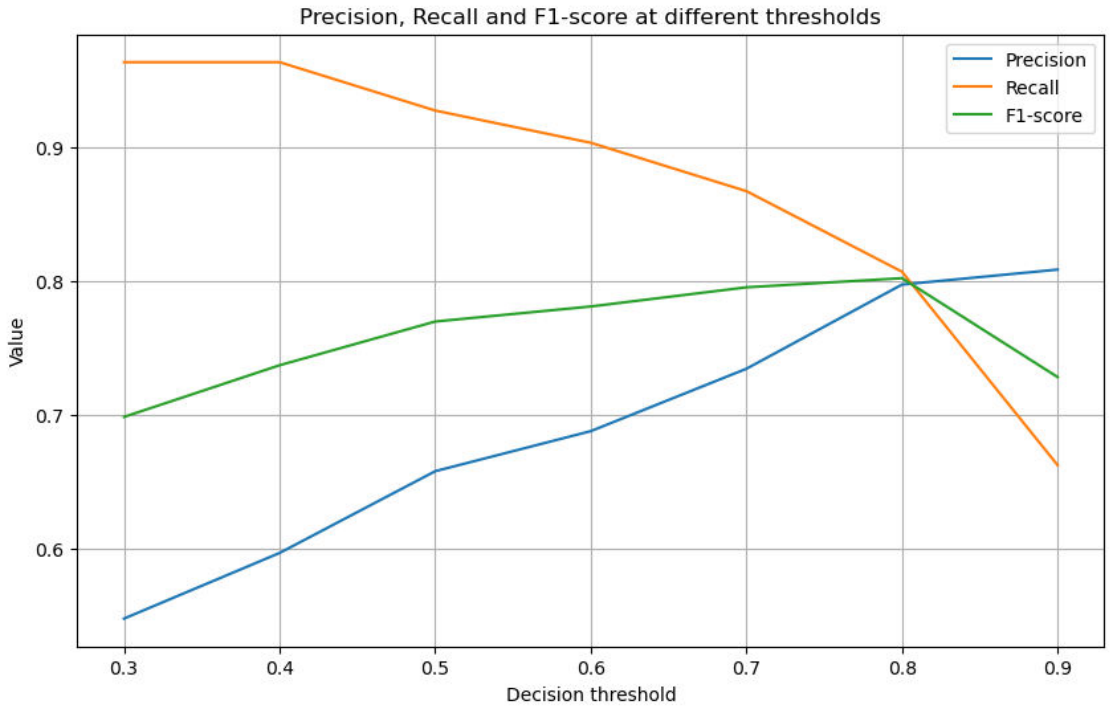
When properly configured with the **class_weight='balanced'** option, Logistic Regression can handle unbalanced data. However, if the classes are very imbalanced, the model may be biased towards the majority class. It works well for moderate imbalance, but **may need additional techniques to deal with a large imbalance**.

```
# Suppress all warnings globally.
warnings.filterwarnings("ignore")

# First let's see the distribution of classes.
print("Distribution of classes in y_train:", np.bincount(y_train))

# Standardization of input data.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# Base model creation and evaluation.
def evaluate_model(y_true, y_pred):
    precision = precision_score(y_true, y_pred)
```

```
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    accuracy = accuracy_score(y_true, y_pred)

    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
    print(f"Accuracy: {accuracy:.4f}")

    return precision, recall, f1, accuracy

# Setting parameters with GridSearchCV.
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'class_weight': ['balanced', {0:1, 1:6}, {0:1, 1:8}, {0:1, 1:10}],
    'solver': ['liblinear', 'saga'],
    'max_iter': [1000],
    'penalty': ['l1', 'l2']
}

# Create a base model.
base_model = LogisticRegression(random_state=42)

# GridSearchCV setup with focus on F1-score.
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(
    estimator=base_model,
    param_grid=param_grid,
    cv=skf,
    scoring='f1',  # We change to F1-score.
    n_jobs=-1,
    verbose=1
)

# Fulfillment of the search.
print("The search for optimal parameters begins...")
grid_search.fit(X_train_scaled, y_train)

# Displaying the results.
print("\nBest parameters:", grid_search.best_params_)
print("Best F1 score:", grid_search.best_score_)

# Using the best model.
best_model = grid_search.best_estimator_

# Model estimation with cross-validation.
print("\nEstimation of the best model with cross-validation:")
y_pred = cross_val_predict(best_model, X_train_scaled, y_train, cv=skf)
precision, recall, f1, accuracy = evaluate_model(y_train, y_pred)

# Experiment with different decision thresholds.
print("\nExperimenting with different decision thresholds:")
y_pred_proba = cross_val_predict(best_model, X_train_scaled, y_train, cv=skf, m
ethod='predict_proba')
```

```
thresholds = [0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
results = []

for threshold in thresholds:
    y_pred_threshold = (y_pred_proba[:, 1] >= threshold).astype(int)
    precision = precision_score(y_train, y_pred_threshold)
    recall = recall_score(y_train, y_pred_threshold)
    f1 = f1_score(y_train, y_pred_threshold)
    results.append({
        'threshold': threshold,
        'precision': precision,
        'recall': recall,
        'f1': f1
    })

# Visualization of threshold results.
results_df = pd.DataFrame(results)
plt.figure(figsize=(10, 6))
plt.plot(results_df['threshold'], results_df['precision'], label='Precision')
plt.plot(results_df['threshold'], results_df['recall'], label='Recall')
plt.plot(results_df['threshold'], results_df['f1'], label='F1-score')
plt.xlabel('Decision threshold')
plt.ylabel('Value')
plt.title('Precision, Recall and F1-score at different thresholds')
plt.legend()
plt.grid(True)
plt.show()

# Finding the best F1-score threshold.
best_f1_threshold = results_df.loc[results_df['f1'].idxmax(), 'threshold']
print(f"\nBest threshold for F1-score: {best_f1_threshold}")

# Final predictions with the best threshold.
final_predictions = (y_pred_proba[:, 1] >= best_f1_threshold).astype(int)
print("\nFinal results with optimized threshold:")
evaluate_model(y_train, final_predictions)
```

```
Distribution of classes in y_train: [528  83]
The search for optimal parameters begins...
Fitting 5 folds for each of 96 candidates, totalling 480 fits

Best parameters: {'C': 1, 'class_weight': {0: 1, 1: 6}, 'max_iter': 1000, 'pena
lty': 'l2', 'solver': 'saga'}
Best F1 score: 0.7713315339631129

Estimation of the best model with cross-validation:
Precision: 0.6581
Recall: 0.9277
F1-Score: 0.7700
Accuracy: 0.9247

Experimenting with different decision thresholds:
```




```
Best threshold for F1-score: 0.8

Final results with optimized threshold:
Precision: 0.7976
Recall: 0.8072
F1-Score: 0.8024
Accuracy: 0.9460

(0.7976190476190477,
 0.8072289156626506,
 0.8023952095808383,
 0.9459901800327333)
```

3. Decision Tree

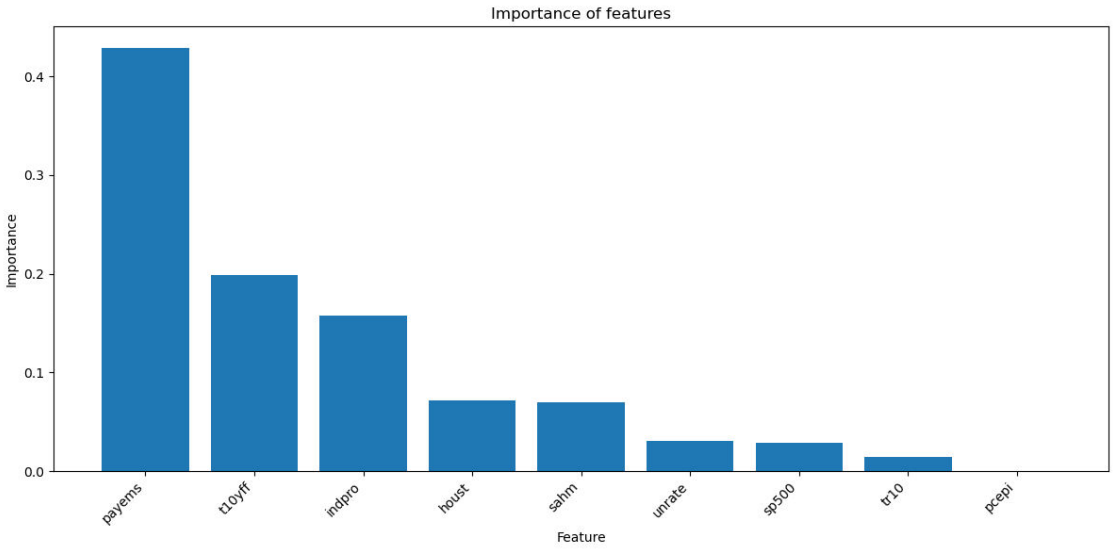
Decision Trees can handle unbalanced data, but without regularization they tend to be biased towards the majority class if the classes are highly imbalanced. **Not the best choice for a large imbalance if no balancing techniques are used.**

```
# List of feature names
feature_names = [
    'sahm',
    'indpro',
    'sp500',
    'tr10',
    't10yff',
    'unrate',
    'pcepi',
    'payems',
    'houst'
]

# First let's see the class distribution.
```

```
print("Distribution of classes in y_train:", np.bincount(y_train))
imbalance_ratio = np.bincount(y_train)[0] / np.bincount(y_train)[1]
print(f"Imbalance ratio (majority to minority): {imbalance_ratio:.2f}")

# Standardization of input data.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

def evaluate_model(y_true, y_pred):
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    accuracy = accuracy_score(y_true, y_pred)

    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
    print(f"Accuracy: {accuracy:.4f}")

    return precision, recall, f1, accuracy

# Setting parameters with GridSearchCV.
param_grid = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy'],
    'max_features': ['sqrt', 'log2', None]
}

# Create a base model.
base_model = DecisionTreeClassifier(random_state=42)

# GridSearchCV setup with focus on F1-score.
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(
    estimator=base_model,
    param_grid=param_grid,
    cv=skf,
    scoring='f1',
    n_jobs=-1,
    verbose=1
)

# Fulfillment of the search.
print("The search for optimal parameters begins...")
grid_search.fit(X_train_scaled, y_train)

# Displaying the results.
print("\nBest parameters:", grid_search.best_params_)
print("Best F1 score:", grid_search.best_score_)

# Using the best model.
best_model = grid_search.best_estimator_
```

```python
# Evaluation of the model with cross-validation
print("\nEstimation of the best model with cross-validation:")
y_pred = cross_val_predict(best_model, X_train_scaled, y_train, cv=skf)
precision, recall, f1, accuracy = evaluate_model(y_train, y_pred)

# Visualization of the importance of named features.
feature_importance = pd.DataFrame({
    'Feature': feature_names,
    'Importance': best_model.feature_importances_
})
feature_importance = feature_importance.sort_values('Importance', ascending=Fal
se)

# Preview
plt.figure(figsize=(12, 6))
plt.bar(feature_importance['Feature'], feature_importance['Importance'])
plt.title('Importance of features')
plt.xlabel('Feature')
plt.ylabel('Importance')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# Output a table with the importance of the characteristics.
print("\nAttribute Importance Table:")
feature_importance['Importance'] = feature_importance['Importance'].round(4)
print(feature_importance.to_string(index=False))
```

```
Distribution of classes in y_train: [528  83]
Imbalance ratio (majority to minority): 6.36
The search for optimal parameters begins...
Fitting 5 folds for each of 270 candidates, totalling 1350 fits

Best parameters: {'criterion': 'entropy', 'max_depth': 5, 'max_features': None,
'min_samples_leaf': 1, 'min_samples_split': 2}
Best F1 score: 0.804327731092437

Estimation of the best model with cross-validation:
Precision: 0.8333
Recall: 0.7831
F1-Score: 0.8075
Accuracy: 0.9493
```




```
Attribute Importance Table:
Feature  Importance
 payems      0.4289
 t10yff      0.1983
 indpro      0.1573
  houst      0.0717
   sahm      0.0696
 unrate      0.0310
  sp500      0.0287
   tr10      0.0146
  pcepi      0.0000
```

4. Random Forest

Random Forest is more robust to unbalanced classes, especially when used with the **class_weight='balanced_subsample'** option, which compensates for the imbalance. This makes it more stable compared to Decision Tree. **Very suitable for unbalanced data if properly set up.**

```
# First let's see the distribution of classes.
print("Distribution of classes in y_train:", np.bincount(y_train))
imbalance_ratio = np.bincount(y_train)[0] / np.bincount(y_train)[1]
print(f"Imbalance ratio (majority to minority): {imbalance_ratio:.2f}")

# Standardization of input data.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# Parameter setting with GridSearchCV for Random Forest.
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, 15, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2'],
    'class_weight': ['balanced', 'balanced_subsample'],
    'bootstrap': [True, False]
}
```

```
# Creating a basic Random Forest model.
base_model = RandomForestClassifier(random_state=42, n_jobs=-1)

# GridSearchCV setup.
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(
    estimator=base_model,
    param_grid=param_grid,
    cv=skf,
    scoring={
        'precision': 'precision',
        'recall': 'recall',
        'f1': 'f1',
        'accuracy': 'accuracy'
    },
    refit='f1',  # We optimize by F1-score.
    n_jobs=-1,
    verbose=1
)

# Fulfillment of the search.
print("The search for optimal parameters begins...")
grid_search.fit(X_train_scaled, y_train)

# Displaying the results.
print("\nBest parameters:", grid_search.best_params_)
print("\nBest results:")
print(f"F1-score: {grid_search.cv_results_['mean_test_f1'][grid_search.best_ind
ex_]:.4f}")
print(f"Precision: {grid_search.cv_results_['mean_test_precision'][grid_search.
best_index_]:.4f}")
print(f"Recall: {grid_search.cv_results_['mean_test_recall'][grid_search.best_i
ndex_]:.4f}")
print(f"Accuracy: {grid_search.cv_results_['mean_test_accuracy'][grid_search.be
st_index_]:.4f}")

print("\nRecommended hyperparameters for subsequent use:")
print("rf_model = RandomForestClassifier(")
for param, value in grid_search.best_params_.items():
    print(f"    {param}={value},")
print("    random_state=42,")
print("    n_jobs=-1")
print(")")
```

```
Distribution of classes in y_train: [528  83]
Imbalance ratio (majority to minority): 6.36
The search for optimal parameters begins...
Fitting 5 folds for each of 864 candidates, totalling 4320 fits

Best parameters: {'bootstrap': False, 'class_weight': 'balanced', 'max_depth':
15, 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_
estimators': 300}

Best results:
```

```
F1-score: 0.8842
Precision: 0.8926
Recall: 0.8787
Accuracy: 0.9689

Recommended hyperparameters for subsequent use:
rf_model = RandomForestClassifier(
    bootstrap=False,
    class_weight=balanced,
    max_depth=15,
    max_features=sqrt,
    min_samples_leaf=4,
    min_samples_split=10,
    n_estimators=300,
    random_state=42,
    n_jobs=-1
)
```

Modeling 2

```
# import libraries
import pickle
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# XGBoost
import xgboost as xgb
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer, f1_score, recall_score, precision_scor
e, accuracy_score
# CatBoost
from catboost import CatBoostClassifier
# SVM
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

collected_data = pd.read_csv('data/collected_data.csv', index_col=0) # CSV file
data = collected_data.copy()
# Setting column 'date' as index.
data = data.set_index('date', drop=True)
del collected_data

data
```

```
            sahm    indpro     sp500      tr10    t10yff  unrate  pcepi  \
date
1962-02-01 -0.17  0.016229  0.016139 -0.043737  1.650556   0.055  0.042
1962-03-01 -0.17  0.005350 -0.005878 -0.108990  1.078182   0.056  0.021
1962-04-01 -0.10  0.002130 -0.063973 -0.087455  1.043000   0.056  0.018
1962-05-01 -0.07 -0.001066 -0.089914  0.030636  1.441818   0.055  0.010
1962-06-01  0.00 -0.002132 -0.085381  0.035411  1.206667   0.055  0.010
...          ...       ...       ...       ...       ...     ...    ...
2024-05-01  0.37  0.006954  0.046904 -0.056818 -0.847727   0.040 -0.010
```

```
2024-06-01  0.43  0.001500  0.034082 -0.177010 -1.024737   0.041  0.145
2024-07-01  0.53 -0.006214  0.011258 -0.056627 -1.081364   0.043  0.202
2024-08-01  0.57  0.003373  0.022578 -0.377727 -1.459091   0.042  0.142
2024-09-01  0.50 -0.002832  0.019996 -0.147409 -1.406500   0.041  0.217
```

date	payems	houst	recession
1962-02-01	0.827913	0.396825	0
1962-03-01	0.819544	0.478671	0
1962-04-01	0.829109	0.518849	0
1962-05-01	0.817113	0.498512	0
1962-06-01	0.816714	0.459325	0
...	...	...	...
2024-05-01	0.824685	0.415179	0
2024-06-01	0.820780	0.422123	0
2024-07-01	0.821816	0.388889	0
2024-08-01	0.822414	0.437996	0
2024-09-01	0.826200	0.434524	0

```
[752 rows x 10 columns]
```

```
# get X and y
X = data.drop(['recession'], axis=1)
y = data['recession']

# We define the training period.
X_train, y_train = X.loc["1962-02-01":"2012-12-01"], y.loc["1962-02-01":"2012-12-01"]
# We define the test period.
# X_test, y_test = X.loc["2013-01-01":], y.loc["2013-01-01":]
# y_train.describe()
```

Metrics (repeat from previous file)

Here are the main metrics we can use to get the **final score** for each **model**:

1. **Precision**: It measures the proportion of true positive predictions relative to all positive predictions. It is **important** when you **want to minimize false positives**.Precision = TP/(TP+FP)

2. **Recall** (Sensitivity): It measures the proportion of true positive predictions to all actual positive cases. It is useful when you want to minimize false negatives.Recall = TP/(TP+FN)

3. **F1-score**: Is a metric that balances precision and recall. It is calculated as the harmonic mean of precision and recall. F1 Score is useful when seeking a balance between high precision and high recall, as it penalizes extreme negative values of either component.F1 = 2*Precision*Recall/(Precision+Recall)

4. **Confusion Matrix**: Visualizes true and predicted classes, which can help better understand model performance.

5. **Specificity**: It measures the proportion of true negative predictions relative to all actual negative cases. It is useful when you want to **minimize false positives**.Specificity = TN/(TN+FP)

6. **Accuracy**: It measures the proportion of correctly predicted cases (both positive and negative) relative to the total number of cases. Accuracy **can be misleading with unbalanced data**, as it can be high even if the model does not predict the small class well.

5. XGBoost

XGBoost has the **scale_pos_weight** parameter which can correct the imbalance between classes. This makes it one of the best models for unbalanced data. One of the most suitable models **to deal with severe imbalance**.

```
# Define search parameters.
param_grid = {
    'max_depth': [3, 4, 5, 6],
    'learning_rate': [0.01, 0.05, 0.1],
    'n_estimators': [100, 200, 300],
    'min_child_weight': [1, 3, 5],
    'gamma': [0, 0.1, 0.2],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9, 1.0],
    'scale_pos_weight': [1, 3, 5]  # To balance classes.
}

# Building an XGBoost classifier.
xgb_classifier = xgb.XGBClassifier(
    objective='binary:logistic',
    random_state=42,
    eval_metric='logloss'
)

# Defining evaluation metrics.
scoring = {
    'f1': make_scorer(f1_score),
    'recall': make_scorer(recall_score),
    'precision': make_scorer(precision_score),
    'accuracy': make_scorer(accuracy_score)
}

# Create a GridSearchCV object.
grid_search = GridSearchCV(
    estimator=xgb_classifier,
    param_grid=param_grid,
    scoring=scoring,
    refit='f1',  # We optimize against F1-score.
    cv=5,
    verbose=0,
    n_jobs=-1
)

# Fulfillment of the search.
grid_search.fit(X_train, y_train)

# Display the best parameters and results.
print("Best parameters:")
for param, value in grid_search.best_params_.items():
    print(f"{param}: {value}")

print("\nBest results:")
print(f"F1 Score: {grid_search.cv_results_['mean_test_f1'][grid_search.best_ind
```

```
ex_]:.4f}")
print(f"Recall: {grid_search.cv_results_['mean_test_recall'][grid_search.best_i
ndex_]:.4f}")
print(f"Precision: {grid_search.cv_results_['mean_test_precision'][grid_search.
best_index_]:.4f}")
print(f"Accuracy: {grid_search.cv_results_['mean_test_accuracy'][grid_search.be
st_index_]:.4f}")
```

```
C:\Users\USER\anaconda3\Lib\site-packages\numpy\ma\core.py:2820: RuntimeWarning
: invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,

Best parameters:
colsample_bytree: 0.9
gamma: 0.1
learning_rate: 0.05
max_depth: 6
min_child_weight: 3
n_estimators: 100
scale_pos_weight: 1
subsample: 1.0

Best results:
F1 Score: 0.8557
Recall: 0.8088
Precision: 0.9267
Accuracy: 0.9657
```

6. CatBoost

CatBoost also supports imbalance correction parameters (**class_weights**) and can automatically detect imbalance in data. Good **for dealing with unbalanced classes**.

```
# Define search parameters.
param_grid = {
    'learning_rate': [0.01, 0.05, 0.1],
    'depth': [4, 6, 8],
    'iterations': [100, 200, 300],
    'l2_leaf_reg': [1, 3, 5, 7],
    'border_count': [32, 64, 128],
    'bagging_temperature': [0, 1],
    'random_strength': [1, 10],
    # Automatic class balancing.
    'auto_class_weights': ['Balanced'],
    # You can also use specific weights, for example:
    # 'class_weights': [[1, 2], [1, 3], [1, 4]],
}

# Building a CatBoost classifier.
cat_classifier = CatBoostClassifier(
    eval_metric='F1',
    random_seed=42,
    verbose=False,  # We turn off verbose output.
    thread_count=-1  # Using all available processors.
```

```
)

# Defining evaluation metrics.
scoring = {
    'f1': make_scorer(f1_score),
    'recall': make_scorer(recall_score),
    'precision': make_scorer(precision_score),
    'accuracy': make_scorer(accuracy_score)
}

# Create a GridSearchCV object.
grid_search = GridSearchCV(
    estimator=cat_classifier,
    param_grid=param_grid,
    scoring=scoring,
    refit='f1',  # We optimize against F1-score.
    cv=5,
    verbose=0,
    n_jobs=1  # CatBoost has its own parallelization.
)

# Fulfillment of the search.
grid_search.fit(X_train, y_train)

# Display the best parameters and results.
print("Best parameters:")
for param, value in grid_search.best_params_.items():
    print(f"{param}: {value}")

print("\nBest results:")
print(f"F1 Score: {grid_search.cv_results_['mean_test_f1'][grid_search.best_ind
ex_]:.4f}")
print(f"Recall: {grid_search.cv_results_['mean_test_recall'][grid_search.best_i
ndex_]:.4f}")
print(f"Precision: {grid_search.cv_results_['mean_test_precision'][grid_search.
best_index_]:.4f}")
print(f"Accuracy: {grid_search.cv_results_['mean_test_accuracy'][grid_search.be
st_index_]:.4f}")
```

```
Best parameters:
auto_class_weights: Balanced
bagging_temperature: 0
border_count: 32
depth: 8
iterations: 200
l2_leaf_reg: 1
learning_rate: 0.05
random_strength: 10

Best results:
F1 Score: 0.8675
Recall: 0.9051
Precision: 0.8575
Accuracy: 0.9624
```

```
# Saving the best model to a .pkl file.
with open('data/best_model.pkl', 'wb') as file:
    pickle.dump(grid_search.best_estimator_, file)

print("The model has been saved successfully in the: 'data/best_model.pkl'")
```

```
The model has been saved successfully in the: 'data/best_model.pkl'
```

7. SVM

SVM can handle unbalanced classes by using the **class_weight='balanced'** parameter. However, with highly unbalanced data, SVM may not be the best solution. It works well with moderate imbalance, but **may struggle with more imbalance**.

```
# Data standardization (important for SVM).
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# Define search parameters.
param_grid = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['rbf', 'linear'],
    'gamma': ['scale', 'auto', 0.1, 0.01],
    'class_weight': ['balanced', {0:1, 1:2}, {0:1, 1:3}],  # Various balancing
options.
    'random_state': [42]
}

# Creating an SVM classifier.
svm_classifier = SVC(
    probability=True,  # Required for some metrics.
    max_iter=1000,    # We increase the maximum number of iterations.
)

# Defining evaluation metrics.
scoring = {
    'f1': make_scorer(f1_score),
    'recall': make_scorer(recall_score),
    'precision': make_scorer(precision_score),
    'accuracy': make_scorer(accuracy_score)
}

# Create a GridSearchCV object.
grid_search = GridSearchCV(
    estimator=svm_classifier,
    param_grid=param_grid,
    scoring=scoring,
    refit='f1',      # We optimize against F1-score.
    cv=5,
    verbose=0,
    n_jobs=-1        # Using all available processors.
)

# Fulfillment of the search.
```

```
grid_search.fit(X_train_scaled, y_train)

# Display the best parameters and results.
print("Best parameters:")
for param, value in grid_search.best_params_.items():
    print(f"{param}: {value}")

print("\nBest results:")
print(f"F1 Score: {grid_search.cv_results_['mean_test_f1'][grid_search.best_ind
ex_]:.4f}")
print(f"Recall: {grid_search.cv_results_['mean_test_recall'][grid_search.best_i
ndex_]:.4f}")
print(f"Precision: {grid_search.cv_results_['mean_test_precision'][grid_search.
best_index_]:.4f}")
print(f"Accuracy: {grid_search.cv_results_['mean_test_accuracy'][grid_search.be
st_index_]:.4f}")

# Display additional information about the imbalance in the data.
unique, counts = np.unique(y_train, return_counts=True)
class_distribution = dict(zip(unique, counts))
print("\nDistribution of classes in the training data:")
for class_label, count in class_distribution.items():
    print(f"Class {class_label}: {count} examples ({count/len(y_train)*100:.2f}
%)")
```

```
Best parameters:
C: 10
class_weight: {0: 1, 1: 2}
gamma: 0.01
kernel: rbf
random_state: 42

Best results:
F1 Score: 0.7677
Recall: 0.7978
Precision: 0.7810
Accuracy: 0.9330

Distribution of classes in the training data:
Class 0: 528 examples (86.42%)
Class 1: 83 examples (13.58%)
```

ARIMA

```
# import libraries
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller
from itertools import product
from tqdm import tqdm
from datetime import datetime, timedelta
```

```
collected_data = pd.read_csv('data/collected_data.csv') # CSV file
data = collected_data.copy()
# Setting column 'date' as index.
data = data.set_index('date', drop=True)
del collected_data

data
```

```
            sahm    indpro     sp500      tr10    t10yff  unrate  pcepi  \
date
1962-02-01 -0.17  0.016229  0.016139 -0.043737  1.650556   0.055  0.042
1962-03-01 -0.17  0.005350 -0.005878 -0.108990  1.078182   0.056  0.021
1962-04-01 -0.10  0.002130 -0.063973 -0.087455  1.043000   0.056  0.018
1962-05-01 -0.07 -0.001066 -0.089914  0.030636  1.441818   0.055  0.010
1962-06-01  0.00 -0.002132 -0.085381  0.035411  1.206667   0.055  0.010
...          ...       ...       ...       ...       ...     ...    ...
2024-05-01  0.37  0.006954  0.046904 -0.056818 -0.847727   0.040 -0.010
2024-06-01  0.43  0.001500  0.034082 -0.177010 -1.024737   0.041  0.145
2024-07-01  0.53 -0.006214  0.011258 -0.056627 -1.081364   0.043  0.202
2024-08-01  0.57  0.003373  0.022578 -0.377727 -1.459091   0.042  0.142
2024-09-01  0.50 -0.002832  0.019996 -0.147409 -1.406500   0.041  0.217

              payems     houst  recession
date
1962-02-01  0.827913  0.396825          0
1962-03-01  0.819544  0.478671          0
1962-04-01  0.829109  0.518849          0
1962-05-01  0.817113  0.498512          0
1962-06-01  0.816714  0.459325          0
...              ...       ...        ...
2024-05-01  0.824685  0.415179          0
2024-06-01  0.820780  0.422123          0
2024-07-01  0.821816  0.388889          0
2024-08-01  0.822414  0.437996          0
2024-09-01  0.826200  0.434524          0

[752 rows x 10 columns]
```

```
def find_best_arima_params(series):
    # Stationarity check.
    result = adfuller(series.dropna())
    d = 0 if result[1] < 0.05 else 1
    # Defining parameters for testing.
    p = range(0, 3)
    q = range(0, 3)
    pdq = list(product(p, [d], q))
    # Finding the best model.
    best_aic = np.inf
    best_params = None
    for param in pdq:
        try:
            model = ARIMA(series, order=param)
            results = model.fit()
            if results.aic < best_aic:
```

```
                best_aic = results.aic
                best_params = param
        except:
            continue
    return best_params if best_params is not None else (1,1,1)

def forecast_feature(series, periods=6):
    best_params = find_best_arima_params(series)
    model = ARIMA(series, order=best_params)
    results = model.fit()
    forecast = results.forecast(steps=periods)
    return forecast

# We find the last date with data.
last_date_with_data = pd.to_datetime(data.index[data.notna().any(axis=1)][-1])

# We create a new index for the forecast starting from the next month after the
# last data date.
# Important: We use pd.date_range instead of manually generating dates.
new_dates = pd.date_range(
    start=last_date_with_data + pd.offsets.MonthBegin(1),
    periods=6,
    freq='MS'  # Month Start frequency
)
new_indices = new_dates.strftime("%Y-%m-%d")

# Forecast for each column including 'recession'.
forecasted_data = pd.DataFrame(index=new_indices)
for column in tqdm(data.columns, desc="Column prediction"):
    try:
        # We only use the non-empty data for prediction.
        valid_data = data[column].dropna()
        if len(valid_data) > 0:
            forecasted_values = forecast_feature(valid_data)
            forecasted_data[column] = forecasted_values
        else:
            forecasted_data[column] = np.nan
    except Exception as e:
        print(f"Error predicting a column {column}: {str(e)}")
        forecasted_data[column] = np.nan

# Processing values for 'recession'.
if 'recession' in forecasted_data.columns:
    forecasted_data['recession'] = forecasted_data['recession'].replace([np.inf
, -np.inf], np.nan).fillna(-1)
    forecasted_data['recession'] = forecasted_data['recession'].apply(lambda x:
1 if x >= 0.5 else 0 if x >= 0 else -1).astype(int)

# Processing values for 'recession'.
combined_data = pd.concat([data, forecasted_data])

# We sort the index to make sure all dates are in the correct order.
combined_data = combined_data.sort_index()
```

```python
# Print the results.
print("\nCombined data (last 12 rows):")
print(combined_data.tail(12))

# Save the results.
combined_data.to_csv('data/combined_data.csv')
print("\nThe results are saved in 'combined_data.csv'")

# Checking the predicted data
print("\nForecast data:")
print(forecasted_data)
```

```
Column prediction: 100%|████████████████████████████████████████████████████
████████| 10/10 [00:22<00:00,  2.25s/it]


Combined data (last 12 rows):
                sahm    indpro     sp500      tr10    t10yff    unrate  \
2024-04-01  0.370000 -0.001579 -0.042506  0.330591 -0.790909  0.039000
2024-05-01  0.370000  0.006954  0.046904 -0.056818 -0.847727  0.040000
2024-06-01  0.430000  0.001500  0.034082 -0.177010 -1.024737  0.041000
2024-07-01  0.530000 -0.006214  0.011258 -0.056627 -1.081364  0.043000
2024-08-01  0.570000  0.003373  0.022578 -0.377727 -1.459091  0.042000
2024-09-01  0.500000 -0.002832  0.019996 -0.147409 -1.406500  0.041000
2024-10-01  0.420318  0.000263  0.012590  0.032915 -1.212508  0.041562
2024-11-01  0.371263  0.001896 -0.000628  0.000468 -1.054334  0.042184
2024-12-01  0.365931  0.001896  0.012222 -0.000439 -0.907669  0.042725
2025-01-01  0.369188  0.001896 -0.000270 -0.000439 -0.771676  0.043291
2025-02-01  0.373684  0.001896  0.011875 -0.000439 -0.645579  0.043803
2025-03-01  0.377939  0.001896  0.000068 -0.000439 -0.528658  0.044322

               pcepi    payems     houst  recession
2024-04-01  0.322000  0.820381  0.445933          0
2024-05-01 -0.010000  0.824685  0.415179          0
2024-06-01  0.145000  0.820780  0.422123          0
2024-07-01  0.202000  0.821816  0.388889          0
2024-08-01  0.142000  0.822414  0.437996          0
2024-09-01  0.217000  0.826200  0.434524          0
2024-10-01  0.213706  0.821749  0.429305          0
2024-11-01  0.199241  0.820944  0.434200          0
2024-12-01  0.196893  0.821594  0.430898          0
2025-01-01  0.194642  0.821594  0.435397          0
2025-02-01  0.192485  0.821594  0.432419          0
2025-03-01  0.190417  0.821594  0.436556          0

The results are saved in 'combined_data.csv'

Forecast data:
                sahm    indpro     sp500      tr10    t10yff    unrate  \
2024-10-01  0.420318  0.000263  0.012590  0.032915 -1.212508  0.041562
2024-11-01  0.371263  0.001896 -0.000628  0.000468 -1.054334  0.042184
2024-12-01  0.365931  0.001896  0.012222 -0.000439 -0.907669  0.042725
2025-01-01  0.369188  0.001896 -0.000270 -0.000439 -0.771676  0.043291
```

```
2025-02-01  0.373684  0.001896  0.011875 -0.000439 -0.645579  0.043803
2025-03-01  0.377939  0.001896  0.000068 -0.000439 -0.528658  0.044322

               pcepi    payems     houst  recession
2024-10-01  0.213706  0.821749  0.429305          0
2024-11-01  0.199241  0.820944  0.434200          0
2024-12-01  0.196893  0.821594  0.430898          0
2025-01-01  0.194642  0.821594  0.435397          0
2025-02-01  0.192485  0.821594  0.432419          0
2025-03-01  0.190417  0.821594  0.436556          0
```