Computational Tools for Macroeconometrics

Assignment 1

Introduction

This assignment introduces students to practical and theoretical aspects of macroeconometrics, focusing on forecasting using the FRED-MD dataset. Students will learn to handle macroeconomic data, perform necessary transformations, apply univariate models to predict key economic indicators and to evaluate these forecasts.

The FRED-MD dataset

The FRED-MD dataset is a comprehensive monthly database for macroeconomic research compiled by the Federal Reserve Bank of St. Louis. It features a wide array of economic indicators. The list of economic indicators can be obtained from the paper accompanying the data pdf.

The data can be downloaded here. The page contains all the different vintages of the data.

Let us start to download the `current.csv` file:

```
import pandas as pd

# Load the dataset
df = pd.read_csv('~/Downloads/current.csv')

# Clean the DataFrame by removing the row with transformation codes
df_cleaned = df.drop(index=0)
df_cleaned.reset_index(drop=True, inplace=True)
df_cleaned['sasdate'] = pd.to_datetime(df_cleaned['sasdate'], format='%m/%d/%Y')
df_cleaned
```

	sasdate	RPI	W875RX1	DPCERA3M086SBEA	CMRMTSPLx	RETAILx	INDPRO	IPFPNSS
0	1959-01-01	2583.560	2426.0	15.188	2.766768e+05	18235.77392	21.9665	23.3891
1	1959-02-01	2593.596	2434.8	15.346	2.787140e+05	18369.56308	22.3966	23.7048
2	1959-03-01	2610.396	2452.7	15.491	2.777753e+05	18523.05762	22.7193	23.8483
3	1959-04-01	2627.446	2470.0	15.435	2.833627e+05	18534.46600	23.2032	24.1927
4	1959-05-01	2642.720	2486.4	15.622	2.853072e+05	18679.66354	23.5528	24.3936
...	...	...	...	...	...	...	...	...
776	2023-09-01	19111.748	15741.9	116.594	1.507530e+06	705304.00000	103.2096	101.0935
777	2023-10-01	19145.402	15784.6	116.663	1.505477e+06	703528.00000	102.3722	100.5292

	sasdate	RPI	W875RX1	DPCERA3M086SBEA	CMRMTSPLx	RETAILx	INDPRO	IPFPNSS
778	2023-11-01	19213.108	15859.9	117.127	1.514733e+06	703336.00000	102.6710	100.9362
779	2023-12-01	19251.946	15899.0	117.773	1.530296e+06	706180.00000	102.6715	100.8332
780	2024-01-01	19377.558	15948.8	117.639	NaN	700291.00000	102.5739	100.9984

```
# Extract transformation codes
transformation_codes = df.iloc[0, 1:].to_frame().reset_index()
transformation_codes.columns = ['Series', 'Transformation_Code']
```

The transformation codes map variables to the transformations we must apply to each variable to render them (approximately) stationary. The data frame `transformation_codes` has the variable's name (`Series`) and its transformation (`Transformation_Code`). There are six possible transformations (x_t denotes the variable to which the transformation is to be applied):

- `transformation_code=1`: no trasformation
- `transformation_code=2`: Δx_t
- `transformation_code=3`: $\Delta^2 x_t$
- `transformation_code=4`: $log(x_t)$
- `transformation_code=5`: $\Delta log(x_t)$
- `transformation_code=6`: $\Delta^2 log(x_t)$
- `transformation_code=7`: $\Delta(x_t/x_{t-1} - 1)$

We can apply these transformations using the following code:

```
import numpy as np

# Function to apply transformations based on the transformation code
def apply_transformation(series, code):
    if code == 1:
        # No transformation
        return series
    elif code == 2:
        # First difference
        return series.diff()
    elif code == 3:
        # Second difference
        return series.diff().diff()
    elif code == 4:
        # Log
        return np.log(series)
    elif code == 5:
        # First difference of log
        return np.log(series).diff()
    elif code == 6:
        # Second difference of log
        return np.log(series).diff().diff()
    elif code == 7:
        # Delta (x_t/x_{t-1} - 1)
        return series.pct_change()
```

```
    else:
        raise ValueError("Invalid transformation code")

# Applying the transformations to each column in df_cleaned based on transformation_codes
for series_name, code in transformation_codes.values:
    df_cleaned[series_name] = apply_transformation(df_cleaned[series_name].astype(float), float(code))


df_cleaned = df_cleaned[2:]                                                    (1)
df_cleaned.reset_index(drop=True, inplace=True)                                (2)
df_cleaned.head()
```

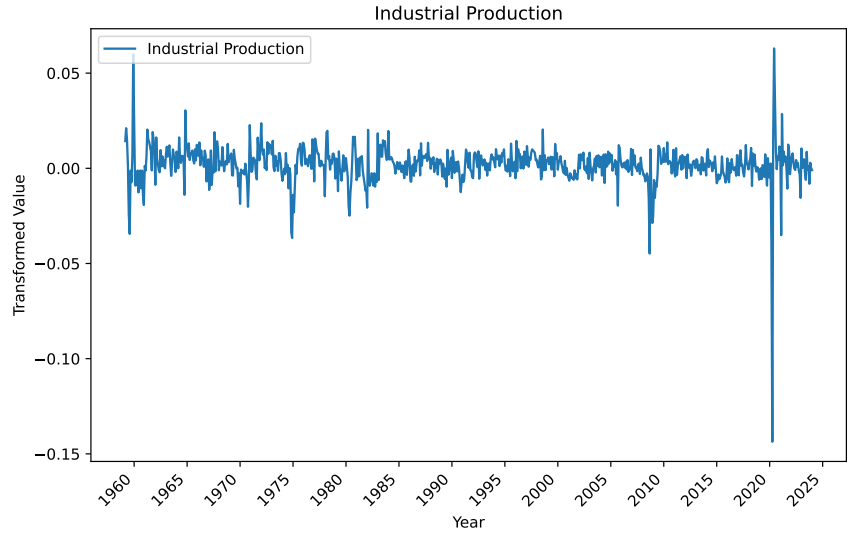
(1) Since some transformations induce missing values, we drop the first two observations of the dataset
(2) We reset the index so that the first observation of the dataset has index 0

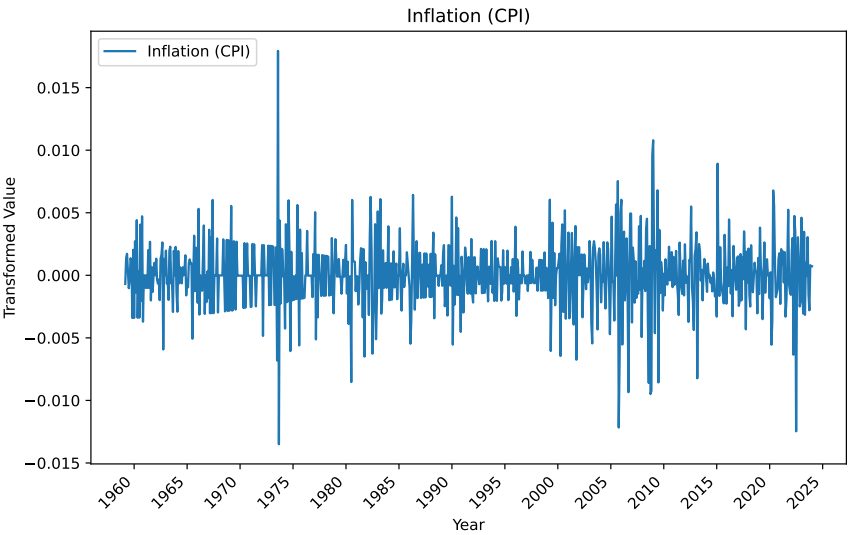
	sasdate	RPI	W875RX1	DPCERA3M086SBEA	CMRMTSPLx	RETAILx	INDPRO	IPFPNSS	IPFI
0	1959-03-01	0.006457	0.007325	0.009404	-0.003374	0.008321	0.014306	0.006035	0.004
1	1959-04-01	0.006510	0.007029	-0.003622	0.019915	0.000616	0.021075	0.014338	0.014
2	1959-05-01	0.005796	0.006618	0.012043	0.006839	0.007803	0.014955	0.008270	0.009
3	1959-06-01	0.003068	0.003012	0.003642	-0.000097	0.009064	0.001141	0.007034	0.007
4	1959-07-01	-0.000580	-0.000762	-0.003386	0.012155	-0.000330	-0.024240	0.001168	0.008

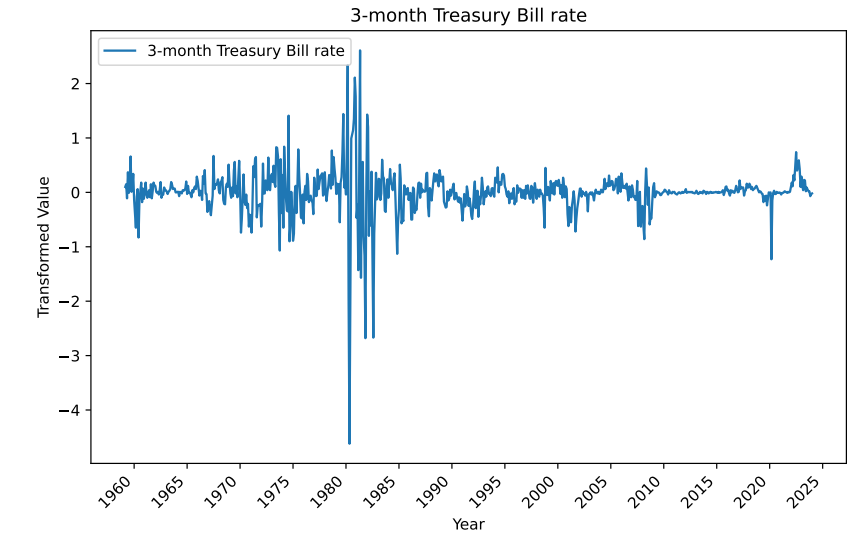
```
import matplotlib.pyplot as plt                                                (1)
import matplotlib.dates as mdates

series_to_plot = ['INDPRO', 'CPIAUCSL', 'TB3MS']                               (2)
series_names = ['Industrial Production',
                'Inflation (CPI)',
                '3-month Treasury Bill rate']


# Create a figure and a grid of subplots
fig, axs = plt.subplots(len(series_to_plot), 1, figsize=(8, 15))               (3)

# Iterate over the selected series and plot each one
for ax, series_name, plot_title in zip(axs, series_to_plot, series_names):
    if series_name in df_cleaned.columns:                                      (4)
        dates = pd.to_datetime(df_cleaned['sasdate'], format='%m/%d/%Y')       (5)
        ax.plot(dates, df_cleaned[series_name], label=plot_title)              (6)
        ax.xaxis.set_major_locator(mdates.YearLocator(base=5))                 (7)
        ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
        ax.set_title(plot_title)                                               (8)
        ax.set_xlabel('Year')                                                  (9)
        ax.set_ylabel('Transformed Value')
        ax.legend(loc='upper left')                                            (10)
        plt.setp(ax.xaxis.get_majorticklabels(), rotation=45, ha='right')      (11)
    else:
        ax.set_visible(False)  # Hide plots for which the data is not available
```

```
27
28  plt.tight_layout()                                                    (12)
29  plt.show()                                                            (13)
```

(1) We use library `matplotlib` to plot
(2) We consider three series (`INDPRO`, `CPIAUCSL`, `TB3MS`) and assign them human-readable names ("Industrial Production", "Inflation (CPI)", "3-month Treasury Bill rate.").
(3) We create a figure with three (`len(series_to_plot)`) subplots arranged vertically. The figure size is 8x15 inches.
(4) We check if the series exists in each series `df_cleaned` DataFrame columns.
(5) We convert the `sasdate` column to `datetime` format (not necessary, since `sasdate` was converter earlier)
(6) We plot each series against the `sasdate` on the corresponding subplot, labeling the plot with its human-readable name.
(7) We format the x-axis to display ticks and label the x-axis with dates taken every five years.
(8) Each subplot is titled with the name of the economic indicator.
(9) We label the x-axis "Year," and the y-axis "Transformed Value," to indicate that the data was transformed before plotting.
(10) A legend is added to the upper left of each subplot for clarity.
(11) We rotate the x-axis labels by 45 degrees to prevent overlap and improve legibility.
(12) `plt.tight_layout()` automatically adjusts subplot parameters to give specified padding and avoid overlap.
(13) `plt.show()` displays the figure with its subplots.


Industrial Production
Industrial Production
Transformed Value
0.05
0.00
−0.05
−0.10
−0.15
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
2025
Year


Inflation (CPI)
Inflation (CPI)
Transformed Value
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
2025
Year


3-month Treasury Bill rate
3-month Treasury Bill rate
Transformed Value
2
1
0
−1
−2
−3
−4
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
2020
2025
Year

Forecasting in Time Series

Forecasting in time series analysis involves using historical data to predict future values. The objective is to model the conditional expectation of a time series based on past observations.

Direct Forecasts

Direct forecasting involves modeling the target variable directly at the desired forecast horizon. Unlike iterative approaches, which forecast one step ahead and then use those forecasts as inputs for subsequent steps, direct forecasting directly models the relationship between past observations and future value.

ARX Models

Autoregressive Moving with predictors (ARX) models are a class of univariate time series models that extend ARMA models by incorporating exogenous (independent) variables. These models are formulated as follows:

$$\begin{aligned} Y_{t+h} &= \alpha + \phi_0 Y_t + \phi_1 Y_{t-1} + \cdots + \phi_p Y_{t-p} + \theta_{0,1} X_{t,1} + \theta_{1,1} X_{t-1,1} + \cdots + \theta_{p,1} X_{t-p,1} + \cdots + \theta_{0,k} X_{t,k} + \cdots + \theta_{p,k} X_{t-p,k} + u_{t+h} \\ &= \alpha + \sum_{i=0}^{p} \phi_i Y_{t-i} + \sum_{j=1}^{k} \sum_{s=0}^{p} \theta_{s,j} X_{t-s,j} + \epsilon_{t+h} \end{aligned} \tag{1}$$

- Y_{t+h}: The target variable at time $t+h$.
- $X_{t,j}$: Predictors (variable $j = 1, \ldots, k$ at time t).
- p number of lags of the target and the predictors.[1]
- ϕ_i, $i = 0, \ldots, p$, and $\theta_{j,s}$, $j = 1, \ldots, k$, $s = 1, \ldots, r$: Parameters of the model.
- ϵ_{t+h}: error term.

For instance, to predict Industrial Prediction using as predictor inflation and the 3-month t-bill, the target variable is `INDPRO`, and the predictors are `CPIAUSL` and `TB3MS`. Notice that the target and the predictors are the transformed variables. Thus, if we use `INDPRO` as the target, we are predicting the log-difference of industrial production, which is a good approximation for its month-to-month percentage change.

By convention, the data ranges from $t = 1, \ldots, T$, where T is the last period, we have data (for the `df_cleaned` dataset, T corresponds to January 2024).

[1] Theoretically, the number of lags for the target variables and the predictors could be different. Here, we consider the simpler case in which both are equal.

Forecasting with ARX

Suppose that we know the parameters of the model for the moment. To obtain a forecast for Y_{T+h}, the h-step ahead forecast, we calculate

$$\begin{aligned}\hat{Y}_{T+h} &= \alpha + \phi_0 Y_T + \phi_1 Y_{T-1} + \cdots + \phi_p Y_{T-p} \\ &\quad + \theta_{0,1} X_{T,1} + \theta_{1,1} X_{T-1,1} + \cdots + \theta_{p,1} X_{T-p,1} \\ &\quad + \cdots + \theta_{0,k} X_{T,k} + \cdots + \theta_{p,k} X_{T-p,k} \\ &= \alpha + \sum_{i=0}^{p} \phi_i Y_{T-i} + \sum_{j=1}^{k} \sum_{s=0}^{p} \theta_{s,j} X_{T-s,j}\end{aligned}$$

While this is conceptually easy, implementing the steps needed to calculate the forecast is insidious, and care must be taken to ensure we are calculating the correct forecast.

To start, it is convenient to rewrite the model in Equation 1 as a linear model

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{u},$$

where β is the vector (of size $1 + (1+p)(1+k)$)

$$\beta = \begin{pmatrix} \alpha \\ \phi_0 \\ \vdots \\ \phi_p \\ \theta_{0,1} \\ \vdots \\ \theta_{p,1} \\ \vdots \\ \theta_{1,k} \\ \vdots \\ \theta_{p,k} \end{pmatrix},$$

$\mathbf{y}$ and $\mathbf{X}$ are respectively given by

$$\mathbf{y} = \begin{pmatrix} y_{p+h+1} \\ y_{p+h+2} \\ \vdots \\ y_T \end{pmatrix}$$

and

$$\mathbf{X} = \begin{pmatrix} 1 & Y_{p+1} & Y_p & \cdots & Y_1 & X_{p+1,1} & X_{p,1} & \cdots & X_{1,1} & \cdots & X_{p+1,k} & X_{p,k} & \cdots & X_{1,k} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \cdots & \vdots & \vdots & & \vdots \\ 1 & Y_{T-h-1} & Y_{T-h-2} & \cdots & Y_{T-h-p-1} & X_{T-h-1,1} & X_{T-h-2,1} & \cdots & X_{T-h-p-1,1} & \cdots & X_{T-h-1,k} & X_{T-h-2,k} & \cdots & X_{T-h-p-1} \\ 1 & Y_{T-h} & Y_{T-h-1} & \cdots & Y_{T-h-p} & X_{T-h,1} & X_{T-h-1,1} & \cdots & X_{T-h-p,1} & \cdots & X_{T-h,k} & X_{T-h-1,k} & \cdots & X_{T-h-p,k} \end{pmatrix}$$

The size of $\mathbf{X}$ is $(T-p-h) \times 1 + (1+k)(1+p)$ and that of $\mathbf{y}$ is $T-h-p$.

The matrix $\mathbf{X}$ can be obtained in the following way:

```
Yraw = df_cleaned['INDPRO']
Xraw = df_cleaned[['CPIAUCSL', 'TB3MS']]

```

```
num_lags  = 4  ## this is p
num_leads = 1  ## this is h
X = pd.DataFrame()
## Add the lagged values of Y
col = 'INDPRO'
for lag in range(0,num_lags+1):
        # Shift each column in the DataFrame and name it with a lag suffix
        X[f'{col}_lag{lag}'] = Yraw.shift(lag)

for col in Xraw.columns:
    for lag in range(0,num_lags+1):
        # Shift each column in the DataFrame and name it with a lag suffix
        X[f'{col}_lag{lag}'] = Xraw[col].shift(lag)
## Add a column on ones (for the intercept)
X.insert(0, 'Ones', np.ones(len(X)))


## X is now a DataFrame
X.head()
```

	Ones	INDPRO_lag0	INDPRO_lag1	INDPRO_lag2	INDPRO_lag3	INDPRO_lag4	CPIAUCSL_lag0	CPIAU
0	1.0	0.014306	NaN	NaN	NaN	NaN	-0.000690	NaN
1	1.0	0.021075	0.014306	NaN	NaN	NaN	0.001380	-0.0006
2	1.0	0.014955	0.021075	0.014306	NaN	NaN	0.001723	0.0013
3	1.0	0.001141	0.014955	0.021075	0.014306	NaN	0.000339	0.0017
4	1.0	-0.024240	0.001141	0.014955	0.021075	0.014306	-0.001034	0.0003

Note that the first p =4 rows of `X` have missing values.

The vector **y** can be similarly created as

```
y = Yraw.shift(-num_leads)
y
```

```
0      0.021075
1      0.014955
2      0.001141
3     -0.024240
4     -0.034465
         ...
774   -0.008147
775    0.002915
776    0.000005
777   -0.000951
778         NaN
Name: INDPRO, Length: 779, dtype: float64
```

The variable `y` has missing values in the last `h` positions (it is not possible to lead the target beyond T).

Notice also that we must keep the last row of `X` for constructing the forecast.

Now we create two `numpy` arrays with the missing values stripped:

```
## Save last row of X (converted to numpy)
X_T = X.iloc[-1:].values
## Subset getting only rows of X and y from p+1 to h-1
## and convert to numpy array
y = y.iloc[num_lags:-num_leads].values
X = X.iloc[num_lags:-num_leads].values
```

```
X_T
```

```
array([[ 1.00000000e+00, -9.51056709e-04,  4.86991246e-06,
         2.91450984e-03, -8.14668061e-03,  9.25729878e-04,
         7.21400503e-04,  7.26467817e-04,  8.11330254e-04,
        -2.79891559e-03, -1.51527417e-03, -2.00000000e-02,
        -3.00000000e-02, -7.00000000e-02,  2.00000000e-02,
         2.00000000e-02]])
```

Now, we have to estimate the parameters and obtain the forecast.

Estimation

The parameters of the model can be estimated by OLS (the OLS estimates the coefficient of the linear projection of Y_{t+h} on its lags and the lags of X_t).

The OLS estimator of β is

$$\hat{\beta} = (X'X)^{-1}X'Y.$$

While this is the formula used to describe the OLS estimator, from a computational poijnt of view is much better to define the estimator as the solution of the set of linear equations:

$$(X'X)\beta = X'Y$$

The function `solve` can be used to solve this linear system of equation.

```
from numpy.linalg import solve
# Solving for the OLS estimator beta: (X'X)^{-1} X'Y
beta_ols = solve(X.T @ X, X.T @ y)

## Produce the One step ahead forecast
## % change month-to-month INDPRO
forecast = X_T@beta_ols*100
forecast
```

```
array([0.08445815])
```

The variable `forecast` contains now the one-step ahead ($h = 1$ forecast) of `INDPRO`. Since `INDPRO` has been transformed in logarithmic differences, we are forecasting the percentage change (and multiplying by 100 gives the forecast in percentage points).

To obtain the h-step ahead forecast, we must repeat all the above steps using a different `h`.

Forecasting Exercise

How good is the forecast that the model is producing? One thing we could do to assess the forecast's quality is to wait for the new data on industrial production and see how big the forecasting error is. However, this evaluation would not be appropriate because we need to evaluate the forecast as if it were repeatedly used to forecast future values of the target variables. To properly assess the model and its ability to forecast `INDPRO`, we must keep producing forecasts and calculating the errors as new data arrive. This procedure would take time as we must wait for many months to have a series of errors that is large enough.

A different approach is to do what is called a Real-time evaluation. A Real-time evaluation procedure consists of putting ourselves in the shoes of a forecaster who has been using the forecasting model for a long time.

In practice, that is what are the steps to follow to do a Real-time evaluation of the model:

0. Set T such that the last observation of `df` coincides with December 1999;
1. Estimate the model using the data up to T
2. Produce $\hat{Y}_{T+1}, \hat{Y}_{T+2}, \dots, \hat{Y}_{T+H}$
3. Since we have the actual data for January, February, ..., we can calculate the forecasting errors of our model
$$\hat{e}_{T+h} = \hat{Y}_{T+h} - Y_{T+h}, \ h = 1, \dots, H.$$
4. Set $T = T + 1$ and do all the steps above.

The process results are a series of forecasting errors we can evaluate using several metrics. The most commonly used is the MSFE, which is defined as

$$MSFE_h = \frac{1}{J} \sum_{j=1}^{J} \hat{e}^2_{T+j+h},$$

where J is the number of errors we collected through our real-time evaluation.

This assignment asks you to perform a real-time evaluation assessment of our simple forecasting model and calculate the MSFE for steps $h = 1, 4, 8$.

As a bonus, we can evaluate different models and see how they perform differently. For instance, you might consider different numbers of lags and/or different variables in the model.

Hint

A sensible way to structure the code for real-time evaluation is to use several functions. For instance, you can define a function that calculates the forecast given the DataFrame.

```
def calculate_forecast(df_cleaned,
                       p = 4,
                       H = [1,4,8],
```

```
                        end_date = '12/1/1999',
                        target = 'INDPRO',
                        xvars = ['CPIAUCSL', 'TB3MS']):
    ## Subset df_cleaned to use only data up to end_date
    rt_df = df_cleaned[df_cleaned['sasdate'] <= pd.Timestamp(end_date)]
    ## Get the actual values of the target at different steps ahead
    Y_actual = []
    for h in H:
        os = pd.Timestamp(end_date) + pd.DateOffset(months=h)
        Y_actual.append(df_cleaned[df_cleaned['sasdate'] == os][target]*100)
        ## Now Y contains the true values at T+H (multiplying * 100)

    Yraw = rt_df[target]
    Xraw = rt_df[xvars]

    X = pd.DataFrame()
    ## Add the lagged values of Y
    for lag in range(0,p):
        # Shift each column in the DataFrame and name it with a lag suffix
        X[f'{target}_lag{lag}'] = Yraw.shift(lag)

    for col in Xraw.columns:
        for lag in range(0,p):
            X[f'{col}_lag{lag}'] = Xraw[col].shift(lag)

    ## Add a column on ones (for the intercept)
    X.insert(0, 'Ones', np.ones(len(X)))

    ## Save the last row of X (converted to a `numpy` array)
    X_T = X.iloc[-1:].values

    ## While the X will be the same, Y needs to be leaded differently
    Yhat = []
    for h in H:
        y_h = Yraw.shift(-h)
        ## Subset getting only rows of X and y from p+1 to h-1
        y = y_h.iloc[p:-h].values
        X_ = X.iloc[p:-h].values
        # Solving for the OLS estimator beta: (X'X)^{-1} X'Y
        beta_ols = solve(X_.T @ X_, X_.T @ y)
        ## Produce the One step ahead forecast
        ## % change month-to-month INDPRO
        Yhat.append(X_T@beta_ols*100)

    ## Now calculate the forecasting error and return

    return np.array(Y_actual) - np.array(Yhat)
```

With this function, you can calculate real-time errors by looping over the `end_date` to ensure you end the loop at the right time.

```python
t0 = pd.Timestamp('12/1/1999')
e = []
T = []
for j in range(0, 10):
    t0 = t0 + pd.DateOffset(months=1)
    print(f'Using data up to {t0}')
    ehat = calculate_forecast(df_cleaned, p = 4, H = [1,4,8], end_date = t0)
    e.append(ehat.flatten())
    T.append(t0)

## Create a pandas DataFrame from the list
edf = pd.DataFrame(e)
## Calculate the RMSFE, that is, the square root of the MSFE
np.sqrt(edf.apply(np.square).mean())
```

```
Using data up to 2000-01-01 00:00:00
Using data up to 2000-02-01 00:00:00
Using data up to 2000-03-01 00:00:00
Using data up to 2000-04-01 00:00:00
Using data up to 2000-05-01 00:00:00
Using data up to 2000-06-01 00:00:00
Using data up to 2000-07-01 00:00:00
Using data up to 2000-08-01 00:00:00
Using data up to 2000-09-01 00:00:00
Using data up to 2000-10-01 00:00:00

0    0.337110
1    0.512690
2    0.624035
dtype: float64
```

You may change the function `calculate_forecast` to output the actual data and the forecast so you can, for instance, construct a plot.