基于VAR模型的多时间序列分析与预测

熊高贤

1. 简介
   1. 概念

向量自回归模型（vector autoregressive model，简称VAR模型）是非结构性方程组模型，用于估计多个变量之间的动态关系。向量自回归模型把系统中每一个内生变量作为系统中所有内生变量的滞后值的函数来构造模型，从而实现了将单变量自回归模型推广到由多元时间序列变量组成的“向量”自回归模型。比如说存在一个系统，系统内有多个变量，VAR 模型分别将每一个变量作为因变量 Y，而系统内所有变量的滞后值作为自变量来建立方程。这样的话，系统内具有多少个变量就能够建立多少个方程式，这些式子就能描述多个变量之间的动态关系。

VAR模型常用于预测相互联系的时间序列系统以及分析随机扰动对变量系统的动态影响，主要应用于宏观经济学。是处理多个相关经济指标的分析与预测中最容易操作的模型之一。

由于向量自回归模型把每个内生变量作为系统中所有内生变量滞后值的函数来构造模型，从而避开了结构建模方法中需要对系统每个内生变量关于所有内生变量滞后值的建模问题。

1. 数据获取与处理
   1. 数据来源

数据集[1]来源于Yash P. Mehra于1994年发表的文章《Wage Growth and the Inflation Process: An Empirical Approach》，用于分析美国的工资-价格方程。数据集包含了1959年第1季度至1989年第3季度的季度数据，主要用于研究美国的经济变量之间的关系，尤其是在工资增长与通货膨胀之间的动态联系。

该数据集包含了以下变量：

rgnp：实际国内生产总值（Real GNP），即调整通货膨胀后的国内生产总值。

pgnp：潜在实际国内生产总值（Potential Real GNP），即在没有经济过热或衰退的情况下，经济所能达到的最大生产水平。

ulc：单位劳动成本（Unit Labor Cost），即每单位产出的劳动成本，通常用来衡量劳动力市场的压力。

gdfco：个人消费支出固定权重平减指数（Fixed Weight Deflator for Personal Consumption Expenditure Excluding Food and Energy），用于调整个人消费支出（不包括食品和能源）的通货膨胀。

gdf：国内生产总值固定权重平减指数（Fixed Weight GNP Deflator），用于调整国内生产总值（GNP）的通货膨胀。

gdfim：进口固定权重平减指数（Fixed Weight Import Deflator），用于调整进口商品的通货膨胀。

gdfcf：个人消费支出中的食品平减指数（Fixed Weight Deflator for Food in Personal Consumption Expenditure），用于衡量食品类消费品的价格变化。

gdfce：个人消费支出中的能源平减指数（Fixed Weight Deflator for Energy in Personal Consumption Expenditure），用于衡量能源类消费品的价格变化。

* 1. 数据初探

首先进行数据可视化，实际GDP从1960年开始逐步上升，表现出稳定的增长趋势。中间有一些微小的波动，可能反映了经济周期的变化。

潜在GDP与实际GDP类似，同样是稳定上升的趋势。这是经济中的潜在生产能力，通常与生产效率和资源利用率相关。

单位劳动成本逐步上升，反映了劳动力成本的持续增长。劳动力成本的上升可能与工资增长和通货膨胀相关。

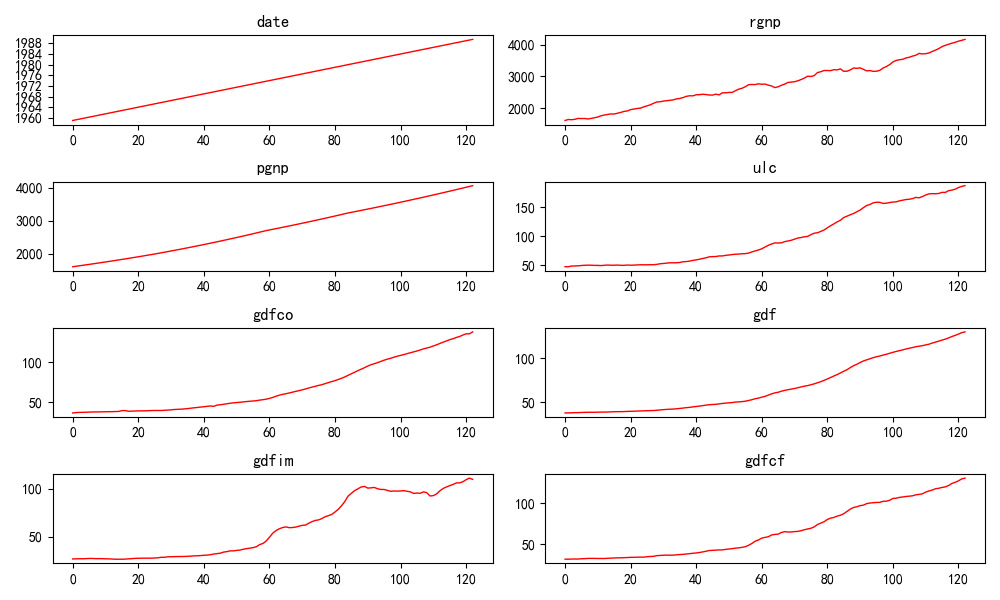
个人消费价格指数显示了个人消费的价格变化，呈现稳步上升趋势。这说明在研究期间内，消费品的价格逐步上涨。

GDP平减指数与其他消费指标类似，也呈现平稳上升趋势，表明经济增长伴随着通货膨胀。

进口价格波动较大，在某些年份（例如1980年代）出现了显著的变化。可能与国际市场价格波动和汇率变化有关。

食品价格也表现出持续上升的趋势。反映了食品领域的通货膨胀或需求的增加。

能源价格呈现波动性较大的趋势，尤其在1970年代和1980年代波动显著。能源价格的波动可能与国际能源危机相关。



载绘制相关热力矩阵，大多数变量之间的相关性接近1，显示了较强的正相关关系。

实际GDP和潜在GDP相关性为0.99，这符合经济理论。单位劳动成本与 GDP平减指数之间相关性非常高，说明劳动成本与价格水平的变动密切相关。

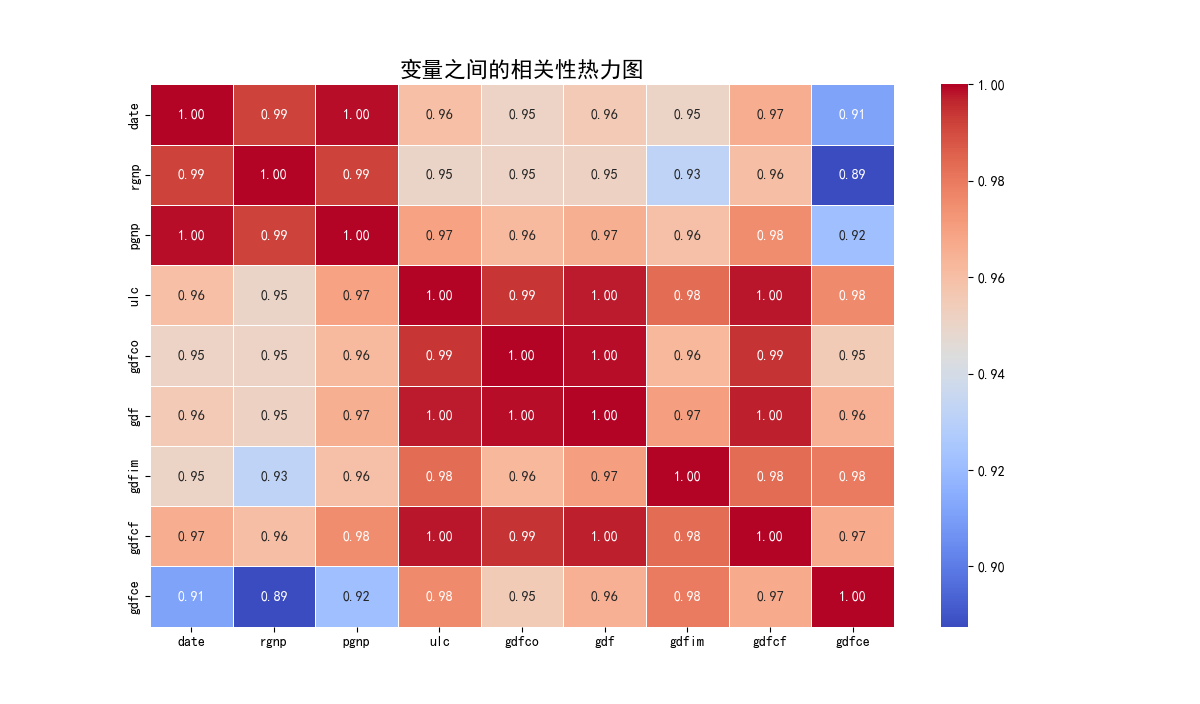
能源价格平减指数与其他变量的相关性稍低，表明能源价格的变化可能有独立的波动因素。

能源和食品之间的相关性为0.97，显示出能源价格可能影响食品价格，这符合现实中能源成本对食品生产的影响。

整体上，大多数变量之间具有高度相关性，反映出它们可能受到共同的经济因素驱动（如通货膨胀、经济增长等）。

能源和食品价格可能具有特殊的驱动因素，需要单独分析其波动原因。

如果需要更深入的因果关系分析，可以采用格兰杰因果检验、构建矢量自回归模型（VAR）进行建模。

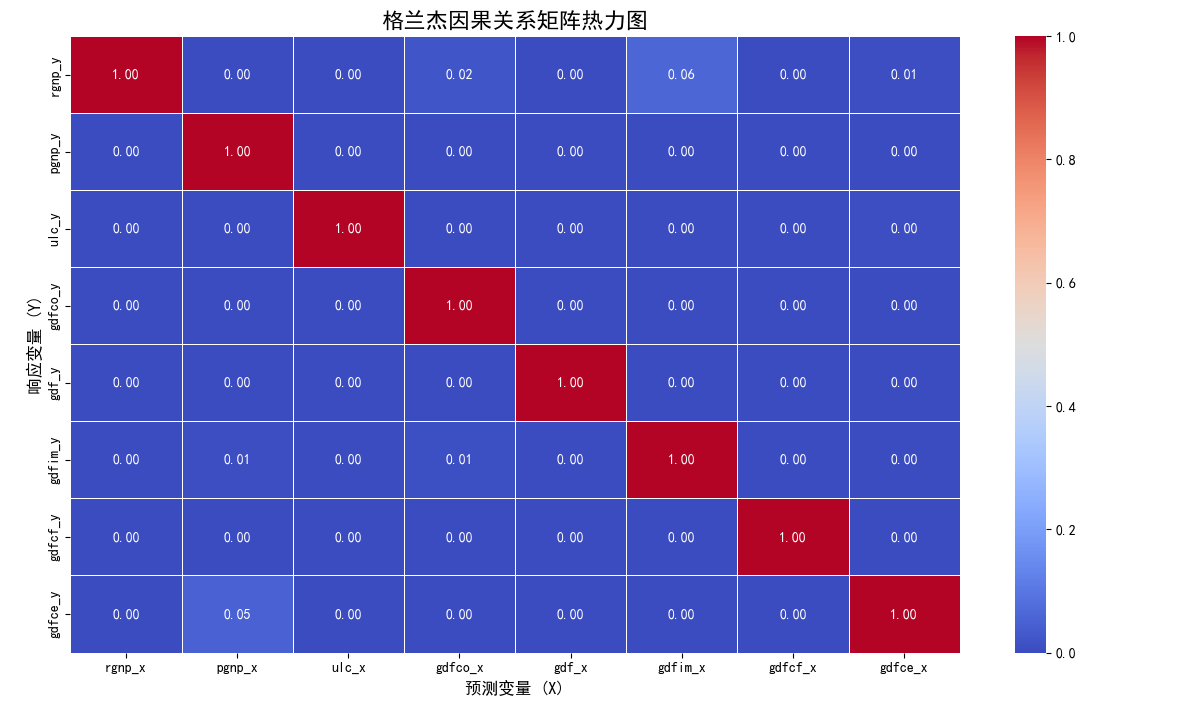


* 1. 格兰杰因果关系

格兰杰因果关系检验是检验时间序列数据中的因果关系的一种方法。它的基本假设是：如果时间序列X在某一时间点的过去值能够帮助预测时间序列Y的未来值，那么我们可以说X"格兰杰导致"Y。

格兰杰因果关系检验的零假设是："变量X不格兰杰导致变量Y"。如果P值小于显著性水平（通常0.05），我们就可以拒绝零假设，即变量X格兰杰导致变量Y。

本数据P值小于0.05可以使用VAR建模。



* 1. ADF检验

对所有数据进行ADF检验，均是非平稳的，因此需要进行协整检验。

* 1. 协整检验

协整检验的目的是检测尽管每个变量本身不平稳，但是否存在某种线性组合使得这些变量变得平稳。如果两个时间序列具有协整关系，就表明它们之间存在长期的、统计显著的关联性。常用的方法包括 Johansen 协整检验、Engle-Granger 协整检验和 Phillips-Ouliaris 协整检验。

绝大部分都满足协整关系，可以进行分析。

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | Test Statistic | Critical Value (95%) | Significant |
| rgnp | 247.99546788882358 | 143.6691 | TRUE |
| pgnp | 183.11714549900563 | 111.7797 | TRUE |
| ulc | 130.01402432631983 | 83.9383 | TRUE |
| gdfco | 85.28482726877719 | 60.0627 | TRUE |
| gdf | 55.05229701347871 | 40.1749 | TRUE |
| gdfim | 31.58883017595778 | 24.2761 | TRUE |
| gdfcf | 14.057176885360148 | 12.3212 | TRUE |
| gdfce | 0.45132 | 4.1296 | FALSE |

* 1. 差分

最后四个时间点作为测试集。

进行差分、在二次差分后均为平稳数据。

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | Difference | P-Value | Significant |
| date | 1st | 0.9662 | FALSE |
| rgnp | 1st | 0 | TRUE |
| pgnp | 1st | 0.3666 | FALSE |
| ulc | 1st | 0.0089 | TRUE |
| gdfco | 1st | 0.5637 | FALSE |
| gdf | 1st | 0.7034 | FALSE |
| gdfim | 1st | 0.0009 | TRUE |
| gdfcf | 1st | 0.2632 | FALSE |
| gdfce | 1st | 0.0228 | TRUE |
| date | 2nd | 0 | TRUE |
| rgnp | 2nd | 0 | TRUE |
| pgnp | 2nd | 0 | TRUE |
| ulc | 2nd | 0 | TRUE |
| gdfco | 2nd | 0 | TRUE |
| gdf | 2nd | 0 | TRUE |
| gdfim | 2nd | 0 | TRUE |
| gdfcf | 2nd | 0 | TRUE |
| gdfce | 2nd | 0 | TRUE |

* 1. 选择模型阶数并训练

选择模型阶数并训练，根据AIC值，lag=4时达到局部最优

|  |  |
| --- | --- |
| Lag Order | AIC |
| 1 | -1.36794 |
| 2 | -1.62124 |
| 3 | -1.7658 |
| 4 | -2.00074 |
| 5 | -1.96195 |
| 6 | -2.33034 |
| 7 | -2.59233 |
| 8 | -3.31726 |
| 9 | -4.80476 |

选择lag=4拟合模型，结果见附件1。

使用durbin watson test，检验残差项中是否还存在相关性，这一步的目的是确保模型已经解释了数据中所有的方差和模式。检验值越接近2，说明模型越好

|  |  |
| --- | --- |
| Variable | Durbin-Watson Statistic |
| rgnp | 2.09 |
| pgnp | 2.02 |
| ulc | 2.17 |
| gdfco | 2.05 |
| gdf | 2.25 |
| gdfim | 1.99 |
| gdfcf | 2.2 |
| gdfce | 2.17 |

模型够好了，下一步进行预测，结果见附件2。

* 1. 转化为原始数据

将预测值和实际值比较，预测效果较好。

图示

描述已自动生成

1. 总结
   1. 实验结果

本次实验使用 VAR（向量自回归）模型对多时间序列数据进行建模和预测，并验证模型的效果。预测值与实际值在趋势上基本吻合，但部分时间点可能存在一定误差。

可视化结果显示，模型能够捕捉数据的主要变化趋势，但短期预测更为准确，长期预测误差有所增加。

VAR 模型在多时间序列建模和短期预测中表现出色，能够有效捕捉变量之间的动态关系。对于非平稳时间序列，通过差分处理后，VAR 模型仍能很好地进行建模和预测。

* 1. 不足

长期预测的误差较大，可能是由于序列中的噪声或非线性关系未被充分捕获。模型对参数选择较为敏感，例如滞后阶数的选择会显著影响预测效果。

* 1. 改进

可以尝试结合其他模型（如 LSTM 或 ARIMA）来进一步优化预测效果。对滞后阶数进行更细致的调参，寻找更优的模型配置。

* 1. 意义

本次实验系统地演示了时间序列分析的全过程，涵盖从数据预处理、平稳性处理到建模、预测和评估等环节。通过实验：加深了对 VAR 模型的理解及其在多时间序列建模中的应用。本实验的经验可以推广到更复杂的时间序列数据分析场景中，为实际问题的建模和决策提供有力支持。

1. 附录
   1. 参考文献

[1] 数据集<https://raw.githubusercontent.com/selva86/datasets/master/Raotbl6.csv>

* 1. 代码

|  |
| --- |
| import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  from statsmodels.tsa.stattools import grangercausalitytests  from statsmodels.tsa.stattools import adfuller  import seaborn as sns  from statsmodels.tsa.vector\_ar.vecm import coint\_johansen  from statsmodels.tsa.api import VAR  from statsmodels.stats.stattools import durbin\_watson  # 设置中文显示  plt.rcParams['font.sans-serif'] = ['SimHei']  # 用于显示中文标签  plt.rcParams['axes.unicode\_minus'] = False    # 解决负号显示问题  # 数据导入和预处理  df = pd.read\_csv('./时间序列分析/VAR/data.csv')  # 读取数据  df['date'] = pd.to\_datetime(df['date'])  df.set\_index('date', inplace=True)  # 将日期设为索引  fig, axes = plt.subplots(nrows=4, ncols=2,figsize=(10,6))  for i, ax in enumerate(axes.flatten()):      data = df[df.columns[i]]      ax.plot(data, color='red', linewidth=1)      ax.set\_title(df.columns[i])  plt.tight\_layout()  plt.show()  # 计算相关性矩阵  correlation\_matrix = df.corr()  # 绘制热力图  plt.figure(figsize=(10, 8))  sns.heatmap(correlation\_matrix, annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)  plt.title("变量之间的相关性热力图", fontsize=16)  plt.show()  df\_numeric = df.select\_dtypes(include=['float64', 'int64'])  # 确保数据中只有数值列  # 定义格兰杰因果关系矩阵函数  def grangers\_causation\_matrix(data, variables, maxlag=12, test='ssr\_chi2test', verbose=False):      # 创建一个空的 DataFrame 用于存储结果      df\_result = pd.DataFrame(np.zeros((len(variables), len(variables))), columns=variables, index=variables)        # 遍历变量，逐一进行格兰杰因果关系检验      for c in df\_result.columns:          for r in df\_result.index:              # 对每对变量进行检验              try:                  test\_result = grangercausalitytests(data[[r, c]], maxlag=maxlag, verbose=False)                  p\_values = [round(test\_result[i + 1][0][test][1], 4) for i in range(maxlag)]  # 提取每个滞后阶数的 p 值                  min\_p\_value = np.min(p\_values)  # 选择最小的 p 值                  df\_result.loc[r, c] = min\_p\_value              except Exception as e:                  # 如果检验失败，标记为 NaN                  df\_result.loc[r, c] = np.nan      # 重命名行和列      df\_result.columns = [var + '\_x' for var in variables]      df\_result.index = [var + '\_y' for var in variables]      return df\_result  # 调用函数并生成结果  granger\_matrix = grangers\_causation\_matrix(df\_numeric, variables=df.columns, maxlag=12, test='ssr\_chi2test', verbose=False)  # 打印格兰杰因果关系矩阵结果  print("格兰杰因果关系矩阵:")  print(granger\_matrix)  # 绘制格兰杰因果关系矩阵的热力图  plt.figure(figsize=(10, 8))  sns.heatmap(granger\_matrix.astype(float), annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)  plt.title("格兰杰因果关系矩阵热力图", fontsize=16)  plt.xlabel("预测变量 (X)", fontsize=12)  plt.ylabel("响应变量 (Y)", fontsize=12)  plt.tight\_layout()  plt.show()  # 定义ADF检验函数  def adfuller\_test(series, signif=0.05, name=''):      """      Perform Augmented Dickey-Fuller test and print results.      """      r = adfuller(series.dropna(), autolag='AIC')  # 去除空值以确保ADF检验的正确性      output = {          'test\_statistic': round(r[0], 4),          'pvalue': round(r[1], 4),          'n\_lags': round(r[2], 4),          'n\_obs': r[3]      }      p\_value = output['pvalue']      # 打印ADF检验的结果      print(f'ADF检验: "{name}"')      print('-' \* 40)      print(f'零假设: 数据存在单位根，非平稳')      print(f'显著性水平: {signif}')      print(f'检验统计量: {output["test\_statistic"]}')      print(f'滞后阶数: {output["n\_lags"]}')      print(f'样本数: {output["n\_obs"]}')      for key, val in r[4].items():          print(f'临界值 {key}: {round(val, 3)}')      if p\_value <= signif:          print(f'P值 = {p\_value}，拒绝零假设。数据是平稳的。')      else:          print(f'P值 = {p\_value}，不能拒绝零假设。数据是非平稳的。')      print('\n')  # 对数据集中的每列进行ADF检验  for name, column in df.items():      adfuller\_test(column, name=name)  # 划分训练集和测试集  nobs = 4  # 最后四个时间点作为测试集  df\_train, df\_test = df\_numeric[0:-nobs], df\_numeric[-nobs:]  # 创建一个列表来存储所有ADF检验的结果  adf\_results = []  # 一阶差分ADF检验结果  df\_differenced = df\_train.diff().dropna()  for name, column in df\_differenced.items():      r = adfuller(column.dropna(), autolag='AIC')      adf\_results.append({          "Variable": name,          "Difference": "1st",          "Test Statistic": round(r[0], 4),          "P-Value": round(r[1], 4),          "Lags Used": round(r[2], 4),          "N Observations": r[3],          "Significant": r[1] < 0.05      })  # 二阶差分ADF检验结果  df\_differenced = df\_differenced.diff().dropna()  for name, column in df\_differenced.items():      r = adfuller(column.dropna(), autolag='AIC')      adf\_results.append({          "Variable": name,          "Difference": "2nd",          "Test Statistic": round(r[0], 4),          "P-Value": round(r[1], 4),          "Lags Used": round(r[2], 4),          "N Observations": r[3],          "Significant": r[1] < 0.05      })  # 将结果转换为DataFrame  adf\_results\_df = pd.DataFrame(adf\_results)  # 保存结果为文件  output\_file = './时间序列分析/VAR/adf\_results.csv'  adf\_results\_df.to\_csv(output\_file, index=False)  # 创建VAR模型  model = VAR(df\_differenced)  # 选择 lag=4 拟合 VAR 模型  model\_fitted = model.fit(4)  # 输出模型摘要  print(model\_fitted.summary())  coefficients = model\_fitted.params  # 将系数矩阵保存为 CSV 文件  var\_coefficients\_file = './时间序列分析/VAR/var\_model\_coefficients.csv'  coefficients.to\_csv(var\_coefficients\_file)    # Durbin-Watson 检验，检查残差的相关性  dw\_test = durbin\_watson(model\_fitted.resid)  # 创建一个结果字典，用于保存 Durbin-Watson 检验结果  dw\_results = {      "Variable": [],      "Durbin-Watson Statistic": []  }  # 将 Durbin-Watson 检验结果存入字典  for col, val in zip(df\_differenced.columns, dw\_test):      dw\_results["Variable"].append(col)      dw\_results["Durbin-Watson Statistic"].append(round(val, 2))  # 转换为 DataFrame  dw\_results\_df = pd.DataFrame(dw\_results)  # 保存为文件  output\_file = './时间序列分析/VAR/durbin\_watson\_results.csv'  dw\_results\_df.to\_csv(output\_file, index=False)  # 确定滞后阶数  lag\_order = model\_fitted.k\_ar  # 获取用于预测的输入数据  forecast\_input = df\_differenced.values[-lag\_order:]  # 进行预测，预测步数为测试集大小 (nobs)  fc = model\_fitted.forecast(y=forecast\_input, steps=nobs)  # 将预测结果转换为 DataFrame  df\_forecast = pd.DataFrame(fc, index=df.index[-nobs:], columns=df\_differenced.columns + '\_2d')  # 保存预测结果为 CSV 文件  forecast\_file = './时间序列分析/VAR/var\_model\_forecast.csv'  # df\_forecast.to\_csv(forecast\_file)  # 定义函数将差分后的值还原为原始数据  def invert\_transformation(df\_train, df\_forecast):      """      Invert the differencing transformation to restore the forecast to the original scale.      """      df\_fc = df\_forecast.copy()      columns = df\_train.columns      for col in columns:          # 还原一阶差分          df\_fc[str(col) + '\_1d'] = (df\_train[col].iloc[-1] - df\_train[col].iloc[-2]) + df\_fc[str(col) + '\_2d'].cumsum()          # 还原到原始数据          df\_fc[str(col) + '\_forecast'] = df\_train[col].iloc[-1] + df\_fc[str(col) + '\_1d'].cumsum()      return df\_fc  # 还原预测值到原始数据  df\_results = invert\_transformation(df\_train, df\_forecast)  # 保存预测结果为文件  forecast\_file = './时间序列分析/VAR/var\_model\_forecast\_restored.csv'  df\_results.to\_csv(forecast\_file)  print(f"预测结果已保存到文件：{forecast\_file}")  # 选择需要比较的列  columns\_to\_plot = df\_numeric.columns  # 创建可视化  # 创建可视化，修改布局防止标题和坐标重叠  fig, axes = plt.subplots(nrows=int(len(columns\_to\_plot) / 2), ncols=2, dpi=150)  for i, (col, ax) in enumerate(zip(columns\_to\_plot, axes.flatten())):      df\_results[col + '\_forecast'].plot(legend=True, ax=ax, label="预测值").autoscale(axis='x', tight=True)      df\_test[col].plot(legend=True, ax=ax, label="实际值")      ax.set\_title(col + ": 预测值 vs 实际值", fontsize=10, pad=10)  # 增加 pad 参数防止标题与图表重叠      ax.xaxis.set\_ticks\_position('none')      ax.yaxis.set\_ticks\_position('none')      ax.spines["top"].set\_alpha(0)      ax.tick\_params(labelsize=8)  # 调整子图之间的间距  plt.tight\_layout(h\_pad=2, w\_pad=2)  # 调整子图的垂直和水平间距  plt.show() |