

## Bayesian Estimation and Forecasting of Time Series in Statsmodels

`Statsmodels`, a Python library for statistical and econometric analysis, has traditionally focused on frequentist inference, including in its models for time series data. This Poster illustrates the powerful features for Bayesian inference of time series models that exist in `statsmodels`, with applications to model fitting, forecasting, time series decomposition, data simulation, and impulse response functions.

### Suggestions for reading and using this Poster

Our suggestion for using this Poster is as follows.

1. The most accessible place to start is the [paper](#). It is relatively short, and guides the reader through the main concepts and key code segments.
2. The poster itself is a Jupyter notebook, `Poster.ipynb`, and it can be (a) run to reproduce the analysis and figures in the paper, and (b) copied and modified to apply these tools and concepts to whatever project the reader wishes!
3. For readers who wish to go further, additional resources on Bayesian analysis of time series models in Statsmodels are listed below.

More details about each of these follow:

#### 1. Paper

Included in this repository is a draft of the paper `Bayesian Estimation and Forecasting of Time Series in Statsmodels.pdf` that is forthcoming in the Proceedings of the 21st Python in Science Conference (SciPy 2022). This paper introduces the time series models included in Statsmodels and shows how to estimate their parameters using Bayesian methods. It also briefly describes the relationship to other popular Python libraries for Bayesian inference, including [PyMC](#), [PyStan](#), and [ArviZ](#).

#### 2. Poster

The paper also provides code samples for several applications that include parameter estimation, forecasting, and causal inference. To keep the paper readable, the code included inline is only brief, but the Poster included in this repository, `Poster.ipynb`, is a Jupyter notebook that contains the complete code for performing the Bayesian analysis in the paper.

#### 3. Additional resources for Bayesian analysis of time series models in Statsmodels

- [Autoregressive moving average \(ARMA\) model](#). This Jupyter notebook shows how to estimate the parameters of an `SARIMAX` model (which is already built-in to Statsmodels) using Bayesian methods.
- [Stochastic volatility model](#). This Jupyter notebook shows (a) how to create a custom state space model that represents a stochastic volatility process, (b) how to apply it to an exchange rate time series, and (c) how to then estimate the parameters using Bayesian methods.
- [Time-Varying Parameter Vector Autoregression \(TVP-VAR\) model](#). This Jupyter notebook shows (a) how to create a custom state space model that represents a TVP-VAR model, and (b) how to then estimate the parameters using Bayesian methods.
- [Dynamic stochastic general equilibrium \(DSGE\) economic model](#). This Jupyter notebook shows how to set up and solve a simple Real Business Cycle economic model, and how to estimate its structural parameters using Bayesian methods.

```
In [1]: !pip install arviz
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: arviz in c:\users\wang xiaoqin\appdata\roaming\python\python312\site-packages (0.22.0)
Requirement already satisfied: setuptools>=60.0.0 in c:\programdata\anaconda3\lib\site-packages (from arviz) (75.1.0)
Requirement already satisfied: matplotlib>=3.8 in c:\programdata\anaconda3\lib\site-packages (from arviz) (3.9.2)
Requirement already satisfied: numpy>=1.26.0 in c:\users\wang xiaoqin\appdata\roaming\python\python312\site-packages
(from arviz) (1.26.4)
Requirement already satisfied: scipy>=1.11.0 in c:\programdata\anaconda3\lib\site-packages (from arviz) (1.13.1)
Requirement already satisfied: packaging in c:\programdata\anaconda3\lib\site-packages (from arviz) (24.1)
Requirement already satisfied: pandas>=2.1.0 in c:\programdata\anaconda3\lib\site-packages (from arviz) (2.2.2)
Requirement already satisfied: xarray>=2023.7.0 in c:\users\wang xiaoqin\appdata\roaming\python\python312\site-packages
(from arviz) (2025.9.0)
Requirement already satisfied: h5netcdf>=1.0.2 in c:\users\wang xiaoqin\appdata\roaming\python\python312\site-packages
(from arviz) (1.6.4)
Requirement already satisfied: typing-extensions>=4.1.0 in c:\users\wang xiaoqin\appdata\roaming\python\python312\site-packages
(from arviz) (4.14.1)
Requirement already satisfied: xarray-einstats>=0.3 in c:\users\wang xiaoqin\appdata\roaming\python\python312\site-packages
(from arviz) (0.9.1)
Requirement already satisfied: h5py in c:\programdata\anaconda3\lib\site-packages (from h5netcdf>=1.0.2->arviz) (3.1
1.0)
Requirement already satisfied: contourpy>=1.0.1 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.8->
arviz) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.8->arviz)
(0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.8-
>arviz) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.8-
>arviz) (1.4.4)
Requirement already satisfied: pillow>=8 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.8->arviz)
(10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=3.8->
arviz) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in c:\programdata\anaconda3\lib\site-packages (from matplotlib>=
3.8->arviz) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\programdata\anaconda3\lib\site-packages (from pandas>=2.1.0->arviz)
(2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\programdata\anaconda3\lib\site-packages (from pandas>=2.1.0->arviz)
(2023.3)
Requirement already satisfied: six>=1.5 in c:\programdata\anaconda3\lib\site-packages (from python-dateutil>=2.7->mat
plotlib>=3.8->arviz) (1.16.0)
```

```
In [2]: # Basic scientific libraries
import numpy as np
```

```

import pandas as pd
from scipy import stats
import matplotlib.pyplot as plt

# Statsmodels is the primarily library used here
# for Bayesian inference of time series models
import statsmodels.api as sm

# Output from Markov chain Monte Carlo (MCMC)
# simulations is naturally multidimensional and
# tends to be quite large. The xarray and Arviz
# Libraries make it easy to work with.
import xarray as xr
import arviz as az

# Formatting settings for the notebook
np.set_printoptions(suppress=True, linewidth=120)

```

## Application 1: estimating the parameters of a random walk model with random walk Metropolis-Hastings

Our first application applies one of the simplest time series models - the random walk - to U.S. Industrial Production.

The random walk model is as follows:

$$y_t = y_{t-1} + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma^2)$$

This model is useful for illustrating the key aspects of Bayesian inference using Statsmodels because it is very simple - it only has a single parameter,  $\sigma^2$ .

Before setting up the model, we will read in the data. Here we are using [U.S. Industrial Production](#), taken from the FRED database made available by the Federal Reserve Bank of St. Louis.

```
In [ ]: # from pandas_datareader.data import DataReader
# # Load data and assign the correct date index
# y = DataReader('IPB50001N', 'fred', start='1980', end='2022-04')['IPB50001N'].asfreq('MS')
# y.index.name = 'date'
```

```
# # Plot the data
# y.plot(title='U.S. Industrial production, monthly', figsize=(15, 5));
```

```
In [3]: df = pd.read_csv(r"C:\Users\WANG XIAOQIN\Desktop\ai_job_dataset.csv")

# Ensure posting_date is in datetime format
df['posting_date'] = pd.to_datetime(df['posting_date'])
df = df.set_index('posting_date')

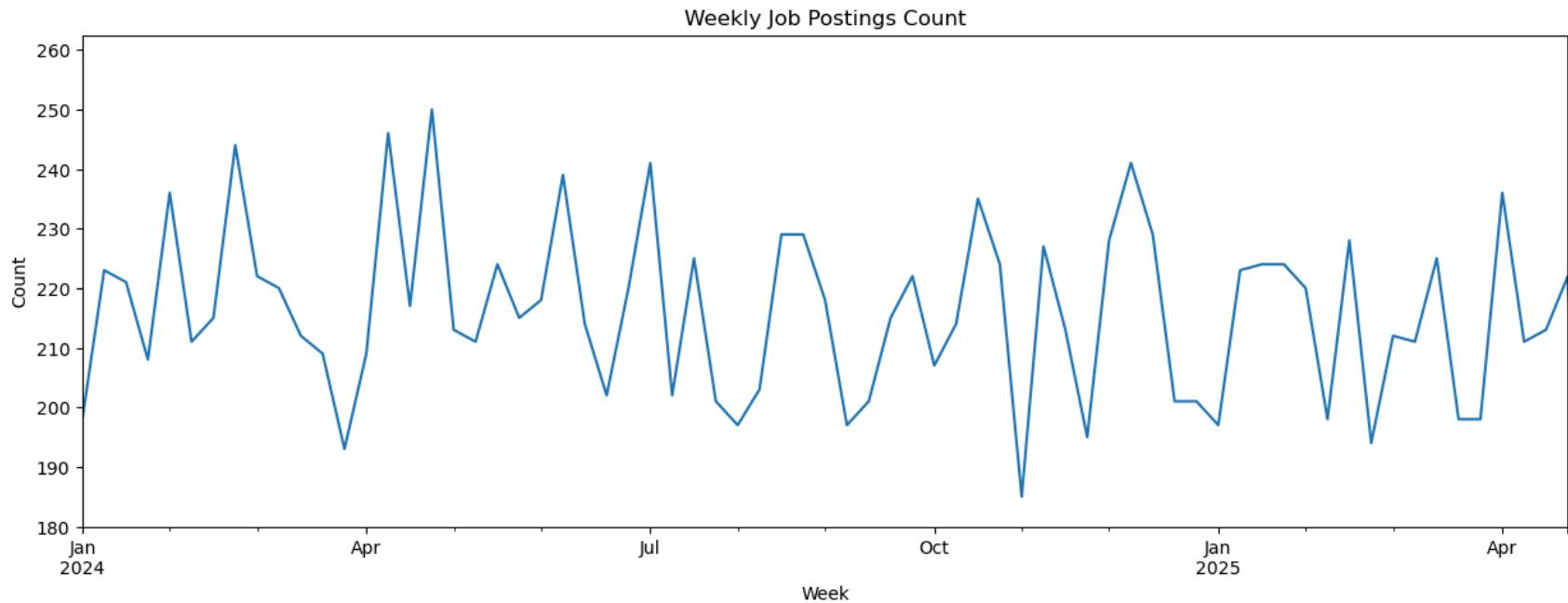
# Aggregate counts by week and assign to y
y = df.resample('W').size()

# Name the index for convenience in later modeling
y.index.name = 'date'
y.name = 'count'
y.head()
```

```
Out[3]: date
2024-01-07    198
2024-01-14    223
2024-01-21    221
2024-01-28    208
2024-02-04    236
Freq: W-SUN, Name: count, dtype: int64
```

```
In [4]: import matplotlib.pyplot as plt

# Plot the data
ax = y.plot(title='Weekly Job Postings Count', figsize=(15, 5))
plt.xlabel('Week')
plt.ylabel('Count')
plt.ylim(180, y.max() * 1.05) # y-axis starts at 180, upper limit slightly higher than the maximum value
plt.xlim(y.index.min(), "2025-04-27") # restrict the range up to 2025-04-27
plt.show()
```



## Constructing a random walk model using `UnobservedComponents`

The `UnobservedComponents` model included in Statsmodels can be used to construct a large number of models that decompose a given time series into trend, seasonal, cyclical, and irregular components (it can also accommodate explanatory variables and autoregressive dynamics). The random walk model shown above is one of the simplest versions.

**Note:** when parameters are estimated with Bayesian methods, the models that can be constructed using `UnobservedComponents` are often referred to as *Bayesian Structural Time Series Models* (BSTS).

```
In [4]: # Construct an object that contains our dataset, `y`,
# and specifies the model, here a random walk
model_rw = sm.tsa.UnobservedComponents(y, 'random walk')

# Check the parameters that the model depends on only
# include a variance term:
print(model_rw.param_names)

['sigma2.level']
```

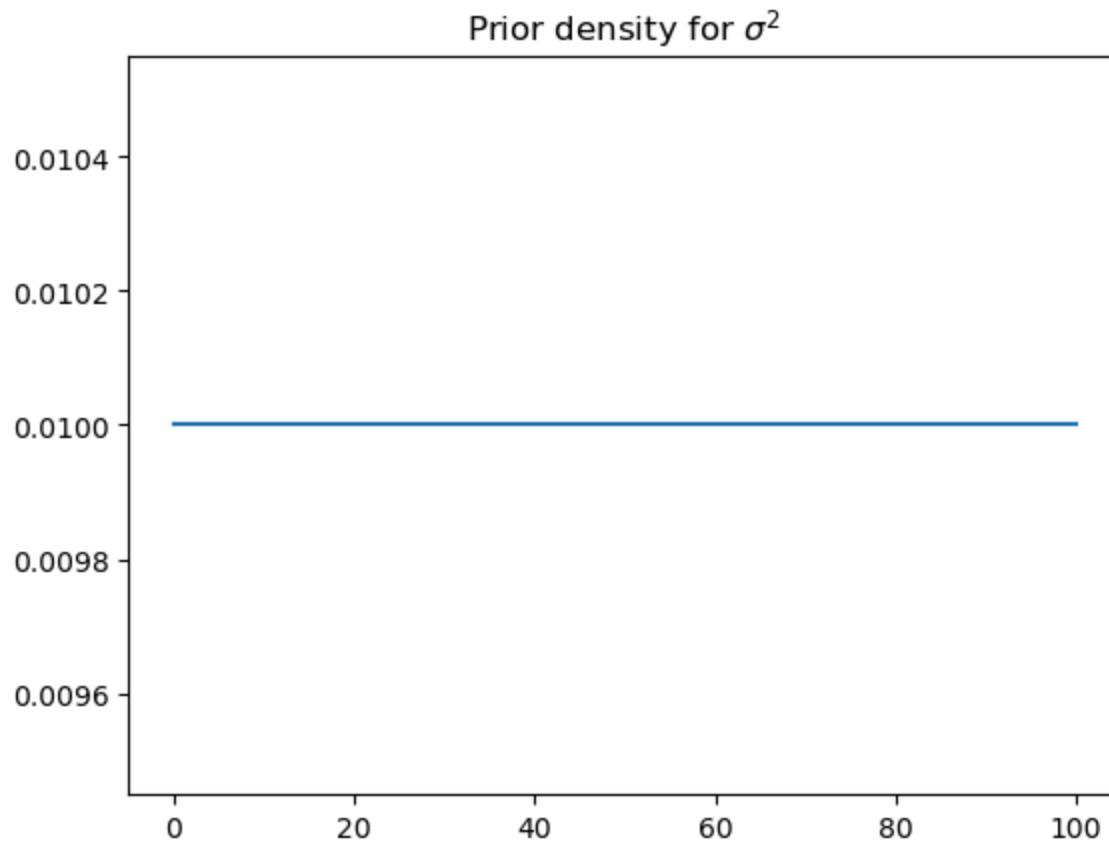
## Estimating $\sigma^2$ using random-walk Metropolis-Hastings (RWMH)

Bayesian inference combines a prior with the likelihood of the data (conditional on the chosen model). Through the model class `UnobservedComponents`, Statsmodels makes available the likelihood function, but we do need to choose our prior.

One benefit of using RWMH is that it does not restrict our choice of prior (unlike Gibbs sampling, described below). To illustrate that flexibility, here we will use a uniform prior, with support over (0.0001, 100).

```
In [5]: # Prior distribution
prior = stats.uniform(0.0001, 100)

# Plot the density
X = np.linspace(0.0001, 100, num=1000)
fig, ax = plt.subplots()
ax.plot(X, prior.pdf(X))
ax.set_title(r'Prior density for $\sigma^2$');
```



Because Random walk Metropolis-Hastings begins each iteration by perturbing the sample from the prior iteration, we also need to specify the perturbation distribution that we will be using. Since we will typically use a Normal distribution centered at zero, the only parameter to select here is the standard deviation. Since this is a simple illustration, we will simply choose the standard deviation to be 0.1.

```
In [6]: # Perturbation distribution  
perturb = stats.norm(scale=0.1)
```

Now we perform the Markov chain Monte Carlo iterations. We first set up the storage arrays for the sampled parameters and then initialize the chain with arbitrary values (however, choosing values that seem likely to lie in higher-density regions of the posterior generally improves the performance of the sampler).

Then, in each iteration, we (1) perturb the value from the previous iteration to compute a new proposed value, (2) compute the probability of accepting the proposed value, and (3) store the proposed value if it is accepted, or else carry over the value from the previous iteration.

```
In [7]: # Storage
niter = 100000
samples_rw = np.zeros(niter + 1)

# Initialization
samples_rw[0] = y.diff().var()
llf = model_rw.loglike(samples_rw[0])
prior_llf = prior.logpdf(samples_rw[0])

# Iterations
for i in range(1, niter + 1):
    # Compute the proposal value
    proposal = samples_rw[i - 1] + perturb.rvs()

    # Compute the acceptance probability
    proposal_llf = model_rw.loglike(proposal)
    proposal_prior_llf = prior.logpdf(proposal)
    accept_prob = np.exp(
        proposal_llf - llf
        + prior_llf - proposal_prior_llf)

    # Accept or reject the value
    if accept_prob > stats.uniform.rvs():
        samples_rw[i] = proposal
        llf = proposal_llf
        prior_llf = proposal_prior_llf
    else:
        samples_rw[i] = samples_rw[i - 1]
```

```
C:\Users\WANG XIAOQIN\AppData\Local\Temp\ipykernel_2204\1617562153.py:19: RuntimeWarning: invalid value encountered in scalar subtract
proposal_llf - llf
```

Although during iterations it is often easiest to store the samples in numpy arrays, for analysis it is more convenient to create an ArviZ `InferenceData` object. We do this below, and then use the ArviZ `plot_posterior` function to plot a histogram of the approximate posterior distribution sampled via RWMH.

Because it takes some time for the Markov chain to converge, we eliminate, or "burn", a number of samples from the beginning of the chain. This is controlled via the `nburn` variable below. In addition, because samples from the Markov chain can be autocorrelated, it is common practice to "thin" the results by using only one out of every so many samples. Here we control the thinning using the `nthin` variable below.

## Application 2: Gibbs sampling with a local level model

In our second application, we consider applying a slightly more general model to Job Postings data. The "local level" model generalizes the random walk model by assuming that the observed data is the sum of a random walk component and an error term. In this way, not all of the movements in the series are passed into the estimates of the trend. To be specific, the model is:

$$\begin{aligned}y_t &= \mu_t + \varepsilon_t, & \varepsilon_t &\sim N(0, \sigma_\varepsilon^2) \\ \mu_t &= \mu_{t-1} + \zeta_t, & \zeta_t &\sim N(0, \sigma_\zeta^2)\end{aligned}$$

This model now has two parameters that we will estimate:  $\sigma_\varepsilon^2$  and  $\sigma_\zeta^2$ .

### Constructing the model in Statsmodels

The local level model can also be constructed using the `UnobservedComponents` class in Statsmodels. Here, we apply the model to job posting counts.

```
In [5]: y.head()
```

```
Out[5]: date
2024-01-07    198
2024-01-14    223
2024-01-21    221
2024-01-28    208
2024-02-04    236
Freq: W-SUN, Name: count, dtype: int64
```

```
In [6]: # weekly job postings series
model_ll = sm.tsa.UnobservedComponents(y, 'local level')
print(model_ll.param_names)

['sigma2.irregular', 'sigma2.level']
```

Here `sigma2.irregular` corresponds to  $\sigma_\varepsilon^2$ , while `sigma2.level` corresponds to  $\sigma_\zeta^2$ .

## Estimating $\sigma_\varepsilon^2, \sigma_\zeta^2$ using Gibbs Sampling

As before, we will need to specify priors for the parameters. However, Gibbs Sampling is only valid when we can construct conditional posterior distributions analytically. In practice, this means that we are limited in the family of priors that we can choose. In this case, we will use [inverse-Gamma priors](#) for each of the variance terms. The inverse-Gamma distribution only has support over strictly positive real numbers, and is often a useful prior for variance parameters. We are still free to choose the hyperparameters of the distribution; here we select values that imply an uninformative, or weak, prior.

```
In [7]: # Specify the prior distributions. With GS, we
# must choose an inverse Gamma prior for each
# variance
priors = [stats.invgamma(0.001, scale=0.001)] * 2
```

### Simulation smoother

When using Gibbs sampling, we do not need to use the accept-reject scheme shown above for sampling parameters. Instead, our choice of prior can be combined with the model likelihood to draw directly from conditional posterior distributions.

In the case of models with unobserved latent states (in the local level model,  $\mu_t$  is such a latent state), we need to be able to draw from the conditional posterior of the latent states (conditional on the data and the variance parameter values). For linear Gaussian state space models (of which the local level model is a special case), such a draw can be constructed with the use of a "simulation smoother".

In `Statsmodels`, a simulation smoother object can be constructed using the `simulation_smooother` method.

**Performance note** the simulation smoother will by default draw more values than we need for Gibbs sampling. To improve performance, we specify the argument `simulate_state=True`, which instructs the simulation smoother to only draw the state vector.

```
In [8]: # Construct the simulation smoother
sim_ll = model_ll.simulation_smoothe(simulate_state=True)
```

## Gibbs sampling iterations

Below we perform the Gibbs sampling iterations. Notice that instead of the accept-reject scheme used with random walk Metropolis-Hastings, we (1) sample from the conditional posterior of the latent states, and then (2) sample from the conditional posteriors of the two variance parameters.

```
In [9]: print(type(y))
```

```
<class 'pandas.core.series.Series'>
```

```
In [10]: # Storage
```

```
niter = 100000
```

```
samples_ll = np.zeros((niter + 1, 2))
```

```
# Initialization
```

```
samples_ll[0] = [y.diff().var(), 1e-5]
```

```
# Iterations
```

```
for i in range(1, niter + 1):
```

```
    # Update the model parameters
```

```
    model_ll.update(samples_ll[i - 1])
```

```
    # Draw from the conditional posterior of the state vector
```

```
    sim_ll.simulate()
```

```
    sample_state = sim_ll.simulated_state.T
```

```
    # Observation error variance
```

```
    resid = y - sample_state[:, 0] # <- 用 .values 转 numpy
```

```
    post_shape = len(resid) / 2 + 0.01
```

```
    post_scale = np.sum(resid**2) / 2 + 0.01
```

```
    samples_ll[i, 0] = stats.invgamma(post_shape, scale=post_scale).rvs()
```

```
    # Level error variance
```

```
    resid = sample_state[1:] - sample_state[:-1]
```

```
    post_shape = len(resid) / 2 + 0.01
```

```
    post_scale = np.sum(resid**2) / 2 + 0.01
```

```
    samples_ll[i, 1] = stats.invgamma(post_shape, scale=post_scale).rvs()
```

As before, we will use ArviZ to visualize the results. Now, since we have two parameters instead of one, we visualize the approximate joint posterior using a hexbin plot.

```
In [11]: # nburn = 10000
# nthin = 10
```

```
# # Convert for use with ArviZ and plot posterior
# samples_ll = az.convert_to_inference_data(
#     {'parameters': samples_ll[None, ...]},
#     coords={'parameter': model_ll.param_names},
#     dims={'parameters': ['parameter']})
# ax = az.plot_pair(samples_ll.posterior.sel(
#     {'draw': np.s_[nburn::nthin]}), kind='hexbin');
```

```
In [13]: # Reinitialize storage arrays before running Gibbs sampling
```

```
niter = 100000
samples_ll = np.zeros((niter + 1, 2)) # make sure this is a numpy array

# Initialize parameters (informative initialization based on the data)
samples_ll[0] = [empirical_irregular_var, empirical_level_var * 0.1] # set the initial value of process variance smc
```

```
NameError
```

```
Traceback (most recent call last)
```

```
Cell In[13], line 6
```

```
    3 samples_ll = np.zeros((niter + 1, 2)) # make sure this is a numpy array
    4 # Initialize parameters (informative initialization based on the data)
----> 6 samples_ll[0] = [empirical_irregular_var, empirical_level_var * 0.1] # set the initial value of process variance smaller
```

```
NameError: name 'empirical_irregular_var' is not defined
```

```
In [14]: # Compute informative prior parameters based on the data
```

```
y_array = y.values
diffs = np.diff(y_array)
empirical_level_var = np.var(diffs)
empirical_irregular_var = np.var(y_array)

# Set prior parameters
alpha = 3
priors = [
    stats.invgamma(alpha, scale=(alpha-1) * empirical_irregular_var),
    stats.invgamma(alpha, scale=(alpha-1) * empirical_level_var)
]
```

```

# Reinitialize storage
niter = 50000 # You can reduce the number of iterations for testing
samples_ll = np.zeros((niter + 1, 2))
samples_ll[0] = [empirical_irregular_var, empirical_level_var * 0.1]

# Initialize the model and simulation smoother
model_ll = sm.tsa.UnobservedComponents(y, 'local level')
sim_ll = model_ll.simulation_smoother(simulate_state=True)

# Gibbs sampling loop
for i in range(1, niter + 1):
    if i % 5000 == 0:
        print(f"Iteration {i}/{niter}")

    # Update model parameters
    model_ll.update(samples_ll[i - 1])

    # Sample latent states
    sim_ll.simulate()
    sample_state = sim_ll.simulated_state.T

    # Sample observation variance sigma2.irregular
    resid = y_array - sample_state[:, 0]
    post_shape = alpha + len(resid) / 2
    post_scale = (alpha-1)*empirical_irregular_var + np.sum(resid**2) / 2
    samples_ll[i, 0] = stats.invgamma(post_shape, scale=post_scale).rvs()

    # Sample process variance sigma2.level
    resid_state = sample_state[1:] - sample_state[:-1]
    post_shape = alpha + len(resid_state) / 2
    post_scale = (alpha-1)*empirical_level_var + np.sum(resid_state**2) / 2
    samples_ll[i, 1] = stats.invgamma(post_shape, scale=post_scale).rvs()

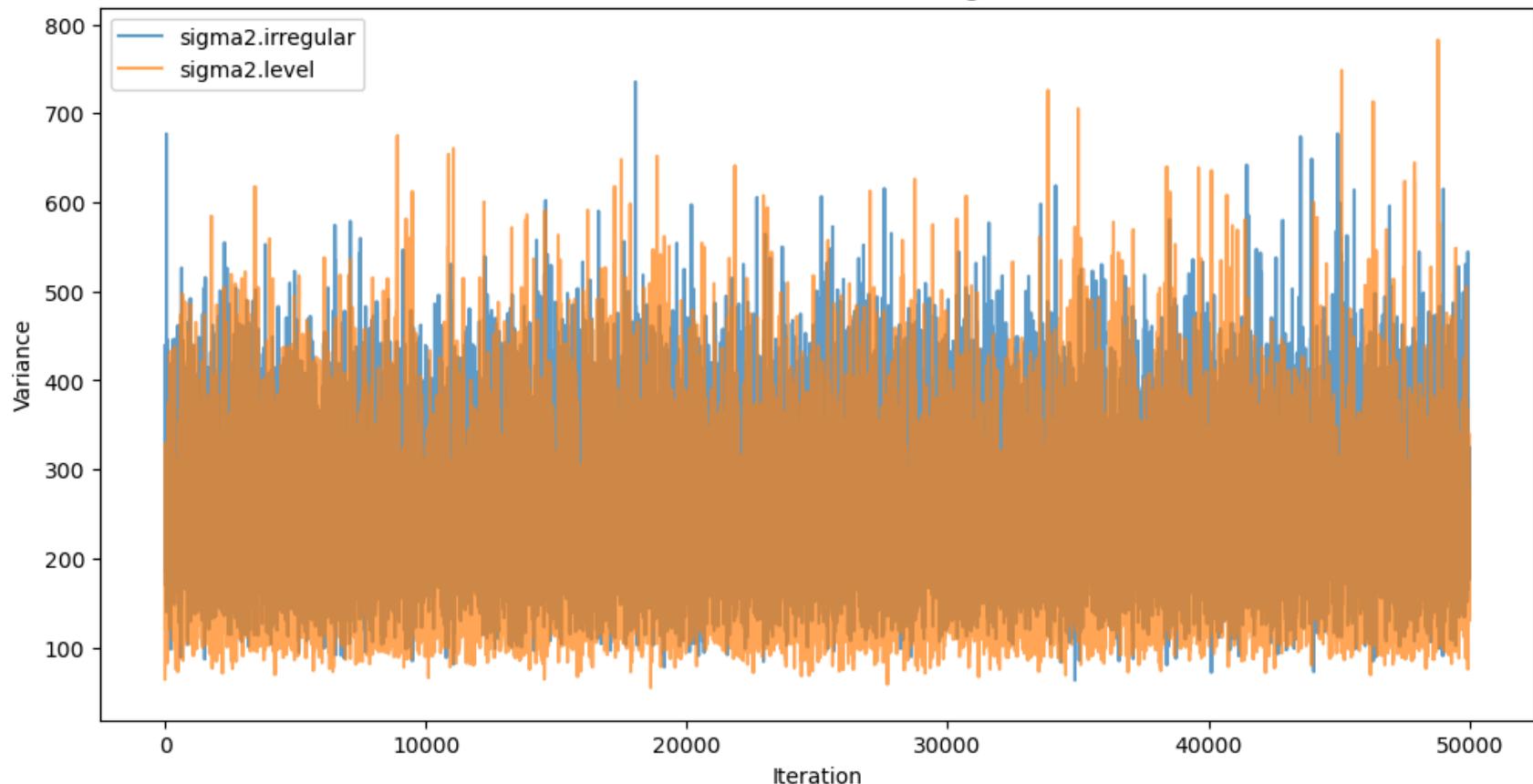
```

```
Iteration 5000/50000
Iteration 10000/50000
Iteration 15000/50000
Iteration 20000/50000
Iteration 25000/50000
Iteration 30000/50000
Iteration 35000/50000
Iteration 40000/50000
Iteration 45000/50000
Iteration 50000/50000
```

```
In [19]: # First check the trace plot to ensure the chain has not collapsed
plt.figure(figsize=(12, 6))
plt.plot(samples_ll[:, 0], label='sigma2.irregular', alpha=0.7)
plt.plot(samples_ll[:, 1], label='sigma2.level', alpha=0.7)
plt.legend()
plt.title('Trace Plot - Check for Convergence')
plt.ylabel('Variance')
plt.xlabel('Iteration')
plt.show()

# Check summary statistics of the last portion of the chain
burnin = niter // 2
print("Last 1000 samples statistics:")
print(f"sigma2.irregular: mean={np.mean(samples_ll[burnin:, 0]):.3f}, std={np.std(samples_ll[burnin:, 0]):.3f}")
print(f"sigma2.level: mean={np.mean(samples_ll[burnin:, 1]):.3f}, std={np.std(samples_ll[burnin:, 1]):.3f}")
```

Trace Plot - Check for Convergence



Last 1000 samples statistics:

sigma2.irregular: mean=257.023, std=70.488  
sigma2.level: mean=212.767, std=74.244

```
In [20]: # Create InferenceData object for formal diagnostics
nburn = 25000 # Discard the first half
nthin = 5      # Apply moderate thinning

samples_clean = samples_ll[nburn::nthin, :]
idata = az.convert_to_inference_data(
    {'parameters': samples_clean[None, ...]},
    coords={'parameter': model_ll.param_names},
    dims={'parameters': ['parameter']}
)
```

```
# Compute key diagnostic statistics
print(az.summary(idata))

arviz - WARNING - Shape validation failed: input_shape: (1, 5001), minimum_shape: (chains=2, draws=4)
          mean      sd   hdi_3%  hdi_97%  mcse_mean  \
parameters[sigma2.irregular] 257.138  70.352 135.780 390.370      1.258
parameters[sigma2.level]     212.742  74.003  95.414 350.327      1.618

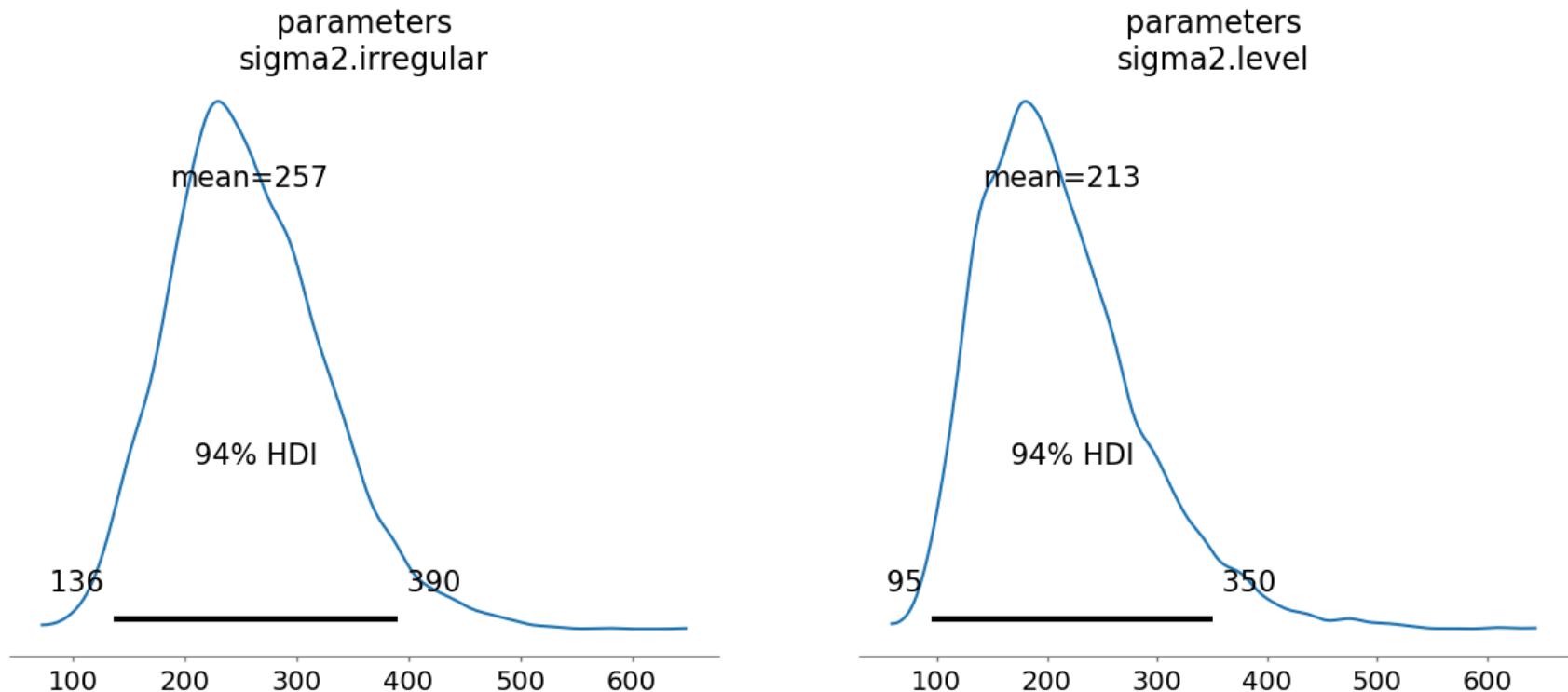
          mcse_sd  ess_bulk  ess_tail  r_hat
parameters[sigma2.irregular]  0.890    2895.0   3643.0   NaN
parameters[sigma2.level]      1.151    2194.0   3266.0   NaN
```

In [21]: `print(idata)`  
`print(idata.posterior)`

```
Inference data with groups:
  > posterior
<xarray.Dataset> Size: 100kB
Dimensions:    (chain: 1, draw: 5001, parameter: 2)
Coordinates:
  * chain      (chain) int32 4B 0
  * draw       (draw)  int32 20kB 0 1 2 3 4 5 ... 4995 4996 4997 4998 4999 5000
  * parameter   (parameter) <U16 128B 'sigma2.irregular' 'sigma2.level'
Data variables:
  parameters  (chain, draw, parameter) float64 80kB 220.9 376.7 ... 197.9
Attributes:
  created_at:    2025-09-24T04:19:34.945218
  arviz_version: 0.17.1
```

In [22]: `import arviz as az`  
`import matplotlib.pyplot as plt`  
`az.plot_posterior(`  
  `idata,`  
  `var_names=["parameters"],`  
  `coords={"parameter": ["sigma2.irregular", "sigma2.level"]},`  
  `point_estimate="mean"`  
`)`

Out[22]: `array([<Axes: title={'center': 'parameters\nsigma2.irregular'}>, <Axes: title={'center': 'parameters\nsigma2.level'}>],`  
`dtype=object)`



The model reasonably decomposes the weekly fluctuations in job postings into observation noise (variance  $256 \pm 68$ ) and true demand variation (variance  $213 \pm 73$ ). The comparable magnitudes of the two indicate a balanced decomposition.

## Extra: Abstracting out the MCMC logic

Although the examples above were very simple, they illustrate the basic principles of random walk Gibbs sampling. We can abstract those operations out into two simple functions that we will use later, to avoid repeating ourselves in each application.

```
In [23]: def draw_mh_step(mod, ix, params, llf, prior_llf, prior, rw, uniform):
    """
    Sample a parameter in RWMH iteration

    Parameters
    -----
    mod : sm.tsa.statespace.mlemodel.MLEModel
```

```
State space model object (for example, in the two
applications above this would have been an instance
of sm.tsa.UnobservedComponents)
ix : int
    Index of the parameter to be sampled within
    the `params` array.
params : array_like
    Current values of the parameters in the Markov
    chain.
llf : float
    Log-likelihood of the data at the given `params`
    array.
prior_llf : float
    Prior log-likelihood of `params[ix]`.
prior : scipy random variable object
    Prior distribution (e.g. the example of this from
    above was `stats.uniform(0.001, 100)`).
rw : stats.norm object
    Perturbation distribution. Should be Normal with
    mean zero.
uniform : stats.uniform object
    Standard uniform distribution object (used for
    choosing whether to accept the proposal)

Returns
-----
draw : float
    Sampled parameter at index `ix`.
accept : bool
    Whether or not the proposal was accepted.
llf : float
    Log-likelihood of the data after accepting or
    rejecting the proposal.
prior_llf : float
    Prior log-likelihood of sampled `draw`.
"""
accept = 0

# Draw the proposal value from the random walk
# proposal distribution
proposal = params.copy()
proposal[ix] += rw.rvs()
```

```

proposal_prior_llf = prior.logpdf(proposal[ix])

# (if the proposal is not in the support of the
# prior, then we definitely won't accept it)
if proposal_prior_llf > -np.inf:

    # Construct the acceptance probability
    proposal_llf = mod.loglike(proposal)
    inner = (proposal_llf - llf +
              proposal_prior_llf - prior_llf)

    if inner > 0:
        acceptance_probability = 1
    else:
        acceptance_probability = np.exp(inner)

    # If we accept the proposal, save the parameter
    # and likelihood values
    if (acceptance_probability == 1
        or acceptance_probability > uniform.rvs()):
        params[ix] = proposal[ix]
        llf = proposal_llf
        prior_llf = proposal_prior_llf
        accept = 1

    # Note: if we didn't accept, then `params[ix]`
    # is unmodified from what was originally passed
    # in to the function

return params[ix], accept, llf, prior_llf

```

In [24]:

```

def draw_ig_gs_step(equation, ix, mod, states, prior):
    """
    Sample a variance parameter in Gibbs sampling iteration

    Assumes a state space model and an inverse-Gamma prior.

    Parameters
    -----
    equation : {'observation', 'state'}
        State space system equation in which the variance
        parameter occurs.

```

```
ix : int or iterable of int
    Index of the variance parameter or parameters to be
    sampled within the given equation. For example, if
    `equation="observation"` and `ix=[0, 1]`, then this
    function samples the variance associated with the
    first column of the data.
mod : sm.tsa.statespace.mlemodel.MLEModel
    State space model object (for example, in the two
    applications above this would have been an instance
    of sm.tsa.UnobservedComponents)
states : array_like
    Current values of the latent state vector in the
    Markov chain.
prior : stats.invgamma object or list of stats.invgamma
    Prior distribution(s) for the variance parameter(s).

>Returns
-----
draw : float or array_like
    Sampled parameter(s) at index `ix` in equation
    `equation`.
"""
# Retrieve the prior hyperparameters
if isinstance(prior, (list, tuple)):
    prior_shape = np.array([p.args[0] for p in prior])
    prior_scale = np.array([p.kwds['scale'] for p in prior])
else:
    prior_shape = prior.args[0]
    prior_scale = prior.kwds['scale']

# Compute the appropriate residual
if equation == 'observation':
    d = mod['obs_intercept', ix]
    Z = mod['design', ix, :]
    resid = mod.endog[:, ix] - d - states @ Z.T
elif equation == 'state':
    c = mod['state_intercept', ix]
    T = mod['transition', ix, :]
    resid = states[1:, ix] - c - states[:-1] @ T.T

squeeze = False
if resid.ndim == 1:
```

```

squeeze = True
resid = resid[:, np.newaxis]

# Draw from the conditional posterior
post_shape = np.sum(~np.isnan(resid), axis=0) / 2 + prior_shape
post_scale = np.nansum(resid**2, axis=0) / 2 + prior_scale
draw = np.array([stats.invgamma(post_shape[i], scale=post_scale[i]).rvs()
                 for i in range(resid.shape[1])])

if squeeze:
    draw = draw[0]

return draw

```

## Application 3: forecasting and seasonal adjustment, with Gibbs Sampling

In our third application, we will show how to augment our Gibbs sampling iterations to forecast job postings. To make the model more realistic, we will also generalize the model still further to allow for trending behavior and to incorporate a seasonal component.

The model is:

$$\begin{aligned}
 y_t &= \mu_t + \gamma_t + \varepsilon_t && \text{observation equation} \\
 \mu_t &= \beta_t + \mu_{t-1} + \zeta_t && \text{level} \\
 \beta_t &= \beta_{t-1} + \xi_t && \text{trend} \\
 \gamma_t &= \gamma_{t-s} + \eta_t && \text{seasonal}
 \end{aligned}$$

The Markov chain Monte Carlo sampling iterations not only produce draws of the model parameters from the approximate posterior distribution, but can also produce draws of (a) the latent states, and (b) forecasts of the observed variable from their respective approximate posterior distributions.

In addition to producing forecasts, these draws allow us to estimate the latent states, including the level, trend and seasonal components. In addition to being interesting in and of themselves, we can use the samples of the seasonal factors to construct a seasonally adjusted version of industrial production.

### Constructing the model in Statsmodels

This model can also be constructed using the `UnobservedComponents` class in Statsmodels, as before. Since we will be using Gibbs sampling, we will also construct the required simulation smoother object (and as described above, we specify `simulate_state=True` to improve performance).

```
In [25]: # Construct the model object...
mod = sm.tsa.UnobservedComponents(y, 'lltrend', seasonal=12, use_exact_diffuse=True)

# ... and simulation smoother object
sim = mod.simulation_smooother(simulate_state=True)

# Print the parameters
print(mod.param_names)

['sigma2.irregular', 'sigma2.level', 'sigma2.trend', 'sigma2.seasonal']
```

## Priors

As shown above, the model now depends on four parameters, each of which are variances. Since we wish to apply Gibbs sampling, we specify inverse Gamma distributions for each, and choose the hyperparameters to imply uninformative priors.

```
In [26]: # Priors
priors = [stats.invgamma(0.001, scale=0.001)] * 4
```

## Forecasting

We now need to set up the details of the forecasting exercise. Here, we choose to forecast for 100 periods, and construct a `DatetimeIndex` that will be associated with those periods.

```
In [27]: # Forecast
n_fcast = 36
ix = pd.date_range(y.index[0], periods=len(y) + n_fcast, freq='W-MON')
fcast_ix = ix[-n_fcast:]
```

## Storage

In the previous applications, we placed the draws from each iteration into numpy arrays, and then later constructed ArviZ objects from those arrays.

Here we'll show a different approach and place the draws from each iteration into xarray objects. These can also be used with ArviZ, as we illustrate afterwards.

```
In [28]: # Storage
niter = 100000
params = xr.DataArray(np.zeros((niter + 1, mod.k_params)),
                      dims=['draw', 'param'],
                      coords={'param': mod.param_names},
                      name='params')
states = xr.DataArray(np.zeros((niter + 1, mod.nobs, mod.k_states)),
                      dims=['draw', 'date', 'state'],
                      coords={'date': mod._index.rename('date'),
                              'state': mod.state_names},
                      name='states')
fcast = xr.DataArray(np.zeros((niter + 1, n_fcast, mod.k_endog)),
                      dims=['draw', 'date', 'variable'],
                      coords={'date': fcast_ix,
                              'variable': [mod.endog_names]},
                      name='forecast')
```

## Gibbs sampling iterations

The iterations here are similar to those from the previous example, except that now we use the function `draw_ig_gs_step` that we defined earlier, to keep the code concise.

```
In [29]: y.head()
```

```
Out[29]: date
2024-01-07    198
2024-01-14    223
2024-01-21    221
2024-01-28    208
2024-02-04    236
Freq: W-SUN, Name: count, dtype: int64
```

```
In [30]: # Initial values
params[0] = np.array([y.var(), 1e-5, 1e-5, 1e-5])

# Iterations
for i in range(1, niter + 1):
```

```

if (i % max(200, niter // 20)) == 0:
    print(i)

# Sample from state posterior
mod.update(params[i - 1])
sim.simulate()
states[i] = sim.simulated_state.T

# Predictive simulation from sample end
fcast[i] = mod.simulate(params[i - 1], n_fcast,
                        initial_state=states[i, -1]).to_frame()

# Gibbs sampling
params[i, 0] = draw_ig_gs_step(
    'observation', 0, mod, sim.simulated_state.T, priors[0])
params[i, 1:4] = draw_ig_gs_step(
    'state', [0, 1, 2], mod, sim.simulated_state.T, priors[1:4])

```

5000  
10000  
15000  
20000  
25000  
30000  
35000  
40000  
45000  
50000  
55000  
60000  
65000  
70000  
75000  
80000  
85000  
90000  
95000  
100000

## Visualizing the approximate posteriors with ArviZ

Because ArviZ accepts xarray objects, we can directly pass our `params` vector to the `plot_posterior` function.

```
In [31]: print(type(params))
print(params.dims)
print(params.coords)
print(params.shape)
```

```
<class 'xarray.core.dataarray.DataArray'>
('draw', 'param')
Coordinates:
 * param    (param) <U16 256B 'sigma2.irregular' ... 'sigma2.seasonal'
(100001, 4)
```

```
In [32]: # nburn = 10000
# nthin = 10
# az.plot_posterior(params.sel(draw=np.s_[nburn + 1::nthin]),
#                     point_estimate='median', kind='hexbin');
```

```
In [33]: import numpy as np
import arviz as az
import matplotlib.pyplot as plt

def plot_posterior_safe(posterior_dict, log=True, bins=50):
    """
    Robust posterior distribution plotting
    -----
    posterior_dict : dict[str, np.ndarray]
        Mapping from parameter name -> sample array
    log : bool
        Whether to take the log of strictly positive parameters
        to avoid variance explosion
    bins : int
        Number of bins for the histogram
    """
    cleaned_dict = {}
    for k, v in posterior_dict.items():
        arr = np.array(v).flatten()
        arr = arr[np.isfinite(arr)] # Remove NaN and Inf
        if arr.size == 0 or np.allclose(arr.min(), arr.max()):
```

```

        continue # Skip invalid arrays
    if log and np.all(arr > 0):
        arr = np.log(arr)
        k = f"log_{k}"
    cleaned_dict[k] = arr

if not cleaned_dict:
    print("⚠️ No valid parameters to plot")
    return

try:
    idata = az.convert_to_inference_data(cleaned_dict)
    az.plot_posterior(
        idata,
        var_names=list(cleaned_dict.keys()),
        point_estimate="median",
        kind="hist",
        bins=bins
    )
    plt.show()
except Exception as e:
    print(f"⚠️ Arviz plotting failed, falling back to Matplotlib: {e}")
    for k, arr in cleaned_dict.items():
        plt.figure()
        plt.hist(arr, bins=bins, density=True, alpha=0.7)
        plt.title(f"Posterior of {k}")
        plt.xlabel(k)
        plt.ylabel("Density")
        plt.show()

```

In [34]: `y.head()`

Out[34]:

date	count
2024-01-07	198
2024-01-14	223
2024-01-21	221
2024-01-28	208
2024-02-04	236

Freq: W-SUN, Name: count, dtype: int64

In [35]: `plot_posterior_safe(posterior_dict, log=True, bins=50)`

```
NameError Traceback (most recent call last)
Cell In[35], line 1
----> 1 plot_posterior_safe(posterior_dict, log=True, bins=50)

NameError: name 'posterior_dict' is not defined
```

## Visualizing the forecasts

The forecasts are also stored in an xarray object. Below we will plot the median forecast along with the 90/10 credible interval.

```
In [41]: y = y.iloc[:-1] # delete final row
print(y)
```

```
date
2024-01-07    198.0
2024-01-14    223.0
2024-01-21    221.0
2024-01-28    208.0
2024-02-04    236.0
...
2025-03-30    198.0
2025-04-06    236.0
2025-04-13    211.0
2025-04-20    213.0
2025-04-27    222.0
Freq: W-SUN, Name: count, Length: 69, dtype: float64
```

```
In [42]: y = y.astype("float64")
print(y.dtype)
```

```
float64
```

```
In [46]: # print(fcast_plot.median(axis=0).to_pandas())
```

```
In [47]: # Check the data without slicing first
fcast_plot = fcast.sel(draw=slice(None))

print("Date range in fcast:", fcast_plot.coords['date'].min().values,
      "to", fcast_plot.coords['date'].max().values)
```

```
# Check the column names
```

```
print(fcast_plot.median(axis=0).to_pandas().head())
```

```
Date range in fcast: 2025-05-12T00:00:00.000000000 to 2026-01-12T00:00:00.000000000
variable      count
date
2025-05-12    135.758330
2025-05-19    227.980047
2025-05-26    187.622895
2025-06-02    199.880458
2025-06-09    198.833553
```

```
In [48]: fig, ax = plt.subplots(figsize=(15, 6))

# Create the full series including both historical and forecast data
last_historical_date = y.index[-1]

# Ensure the forecast starts immediately after the end of historical data
fcast_plot = fcast.sel(
    draw=np.s_[nburn + 1::nthin],
    date=slice(last_historical_date, None) # all forecasts starting from the last historical point
)

forecast_median = fcast_plot.median(axis=0).to_pandas()['count']

# Create a continuous time series
full_timeline = y.append(forecast_median)

# Plot the complete time series in one go
full_timeline.plot(ax=ax, color="#1f77b4", linewidth=2.5, label="Historical + Forecast")

# Use different colors to distinguish between historical and forecast regions
historical_mask = full_timeline.index <= last_historical_date
forecast_mask = full_timeline.index > last_historical_date

# Re-plot the historical part (blue)
ax.plot(full_timeline[historical_mask].index,
        full_timeline[historical_mask].values,
        color="#1f77b4", linewidth=2.5, label="Historical Data")

# Re-plot the forecast part (orange)
ax.plot(full_timeline[forecast_mask].index,
```

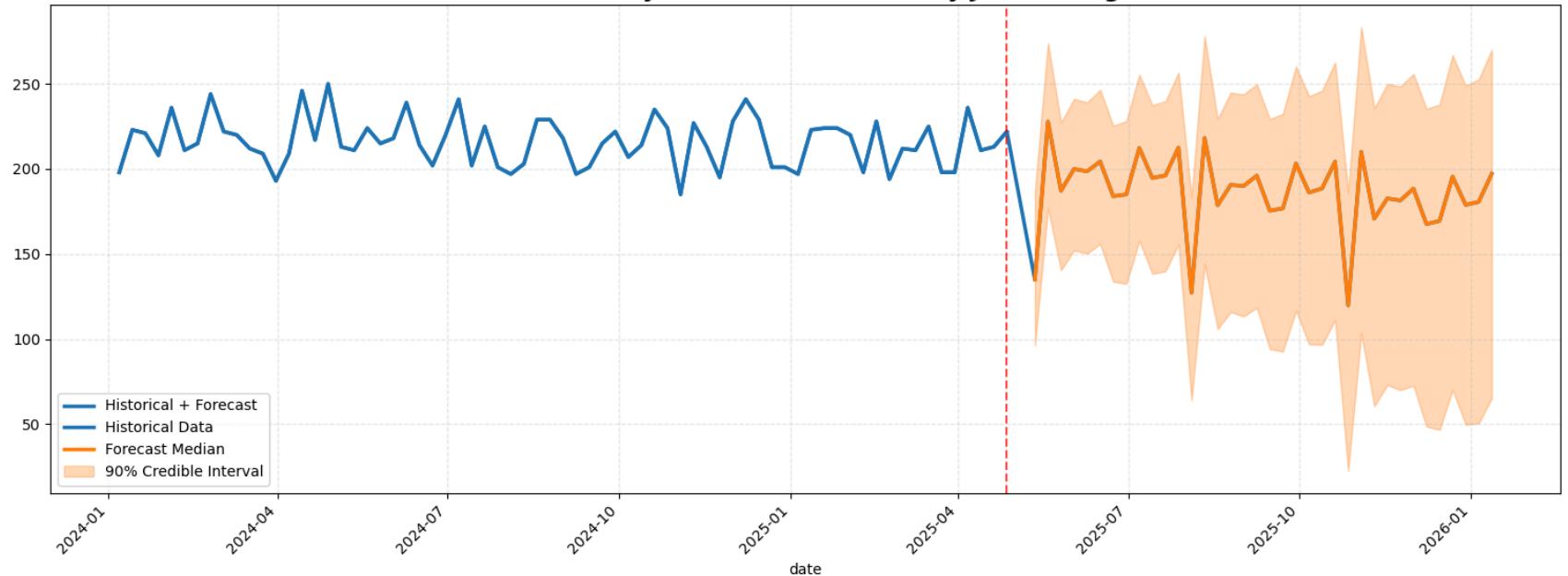
```
full_timeline[forecast_mask].values,
color="#ff7f0e", linewidth=2.5, label="Forecast Median")

# Plot the credible interval (only for the forecast part)
forecast_dates = fcast_plot.coords['date'].to_index()
ax.fill_between(
    forecast_dates,
    fcast_plot.quantile(0.05, dim='draw')[:, 0],
    fcast_plot.quantile(0.95, dim='draw')[:, 0],
    color="#ff7f0e", alpha=0.3, label="90% Credible Interval"
)

# Add a vertical line to separate historical and forecast periods
ax.axvline(x=last_historical_date, color='red', linestyle='--', alpha=0.7)

# Beautify
ax.set_title("Seamless Bayesian Forecast: Weekly Job Postings",
             fontsize=16, fontweight='bold')
ax.legend()
ax.grid(True, linestyle='--', alpha=0.3)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

## Seamless Bayesian Forecast: Weekly Job Postings



### Visualizing the estimated latent states

As noted above, the `states` xarray object contains draws from the approximate posterior of the latent states. Below, we plot the medians and 90/10 credible intervals for the level, trend, and seasonal components.

```
In [50]: fig, axes = plt.subplots(3, figsize=(6, 6));

states_plot = states.sel(draw=np.s_[nburn + 1::nthin])
level = states_plot.sel(state='level')
trend = states_plot.sel(state='trend')
seasonal = states_plot.sel(state='seasonal')

# Plot the estimated Level, over time
level.median(axis=0).to_pandas().rename('Estimated level').plot(ax=axes[0])
axes[0].fill_between(level.coords['date'].to_index(),
                     level.quantile(0.1, dim='draw'),
                     level.quantile(0.9, dim='draw'),
                     color='C0', alpha=0.2)
axes[0].legend(loc=(0, 0.80))
axes[0].set_xlabel(None)
```

```

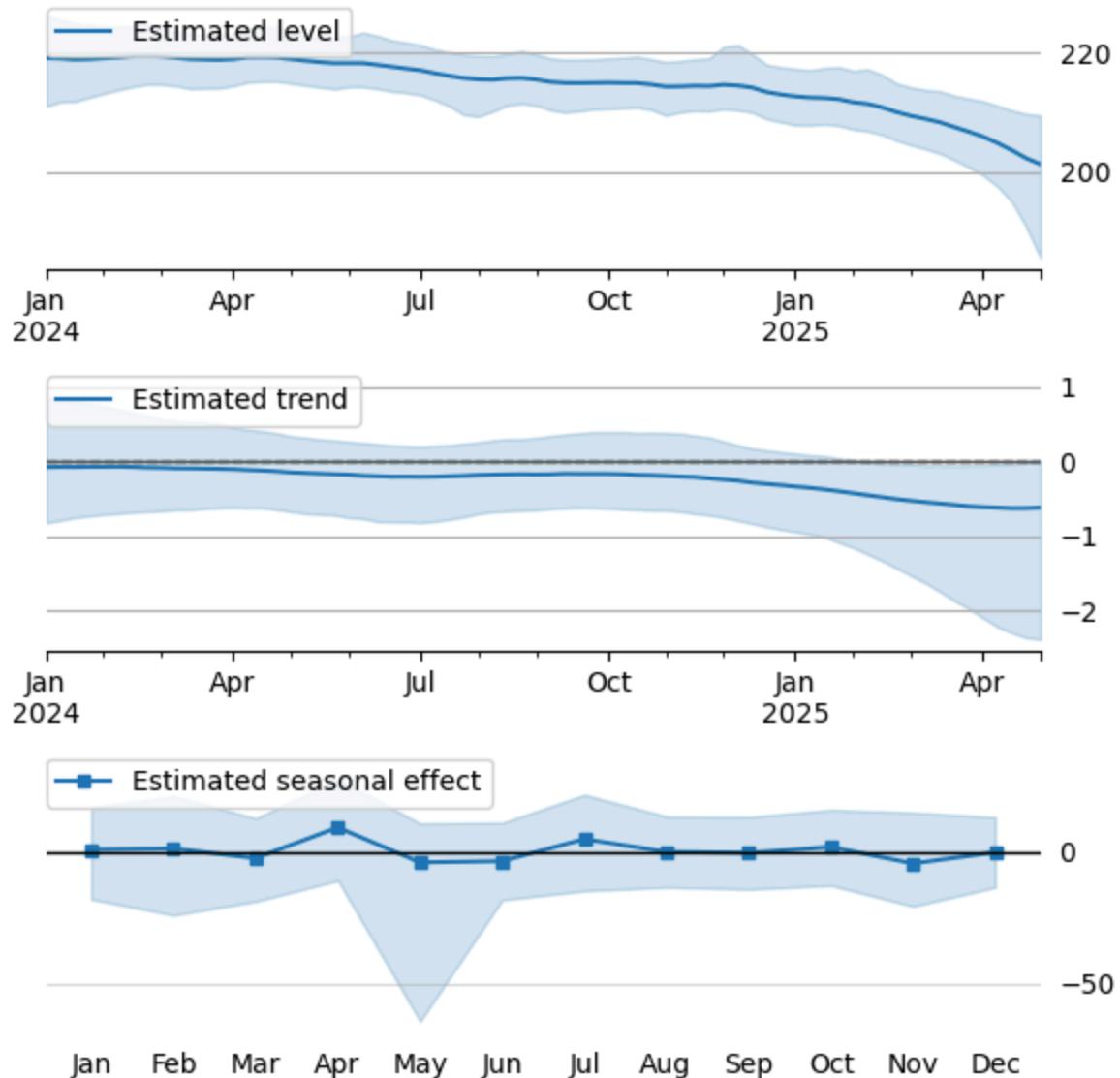
axes[0].yaxis.set_label_coords(1, 1)
axes[0].yaxis.set_tick_params(left=False, right=False, labelleft=False, labelright=True)
axes[0].yaxis.grid()
[axes[0].spines[s].set_visible(False) for s in ['left', 'top', 'right']]

# Plot the estimated trend, over time
trend.median(axis=0).to_pandas().rename('Estimated trend').plot(ax=axes[1])
axes[1].fill_between(trend.coords['date'].to_index(),
                     trend.quantile(0.1, dim='draw'),
                     trend.quantile(0.9, dim='draw'),
                     color='C0', alpha=0.2)
axes[1].axhline(0, color='k', linestyle='--', linewidth=1, zorder=0)
axes[1].legend(loc=(0, 0.85))
axes[1].set_xlabel(None)
axes[1].yaxis.set_label_coords(1, 1)
axes[1].yaxis.set_tick_params(left=False, right=False, labelleft=False, labelright=True)
axes[1].yaxis.grid()
axes[1].axhline(0, color="#555", linewidth=1)
[axes[1].spines[s].set_visible(False) for s in ['left', 'top', 'right']]

# Plot the estimated seasonal components, aggregated over time
x = seasonal.T.to_pandas()
x.index = x.index.month.rename('week')
x = x.T.melt()
g = x.groupby('week')
gm = g.median()['value']
gm.index = np.arange(12)
gm.rename('Estimated seasonal effect').plot(ax=axes[2], marker='s', markersize=5)
axes[2].legend(loc=(0, 0.85))
axes[2].fill_between(np.arange(12),
                     g.quantile(0.1)['value'],
                     g.quantile(0.9)['value'],
                     color='C0', alpha=0.2)
axes[2].xaxis.set_ticks(np.arange(0, 12))
axes[2].xaxis.set_ticklabels(pd.period_range('2024', periods=12, freq='M').strftime('%b'))
axes[2].set_xlabel('')
axes[2].axhline(0, color='k', linewidth=1)
axes[2].yaxis.set_tick_params(left=False, right=False, labelleft=False, labelright=True)
axes[2].yaxis.grid(alpha=0.5)
axes[2].xaxis.set_tick_params(bottom=False)
[axes[2].spines[s].set_visible(False) for s in ['left', 'top', 'right', 'bottom']]

```

```
fig.suptitle('')  
fig.tight_layout()
```



- I applied a Bayesian structural time series model based on Gibbs sampling and successfully decomposed the job posting data into level, trend, and seasonal components. The results indicate that the data are mainly driven by a stable level component

and pronounced seasonal fluctuations, while the long-term trend component is not statistically significant. This confirms that recruitment demand follows a regular monthly cycle but does not exhibit a persistent expansion or contraction pattern.

- The level component reflects the core recruitment demand, while the seasonal component reveals the cyclical patterns of peak and slack periods in hiring activities. The conclusion is that job posting data are better suited for forecasting and planning with a “baseline level + seasonal adjustment” model.

### Constructing a seasonally adjusted index

A seasonally adjusted index  $z_t$  can be constructed by subtracting the estimated seasonal factor from the observed variable:

$$z_t = (y_t - \hat{\gamma}_t) \approx \mu_t + \varepsilon_t$$

```
In [51]: # Construct the seasonally adjusted index
z = (-seasonal.to_pandas().T).add(y, axis=0)
```

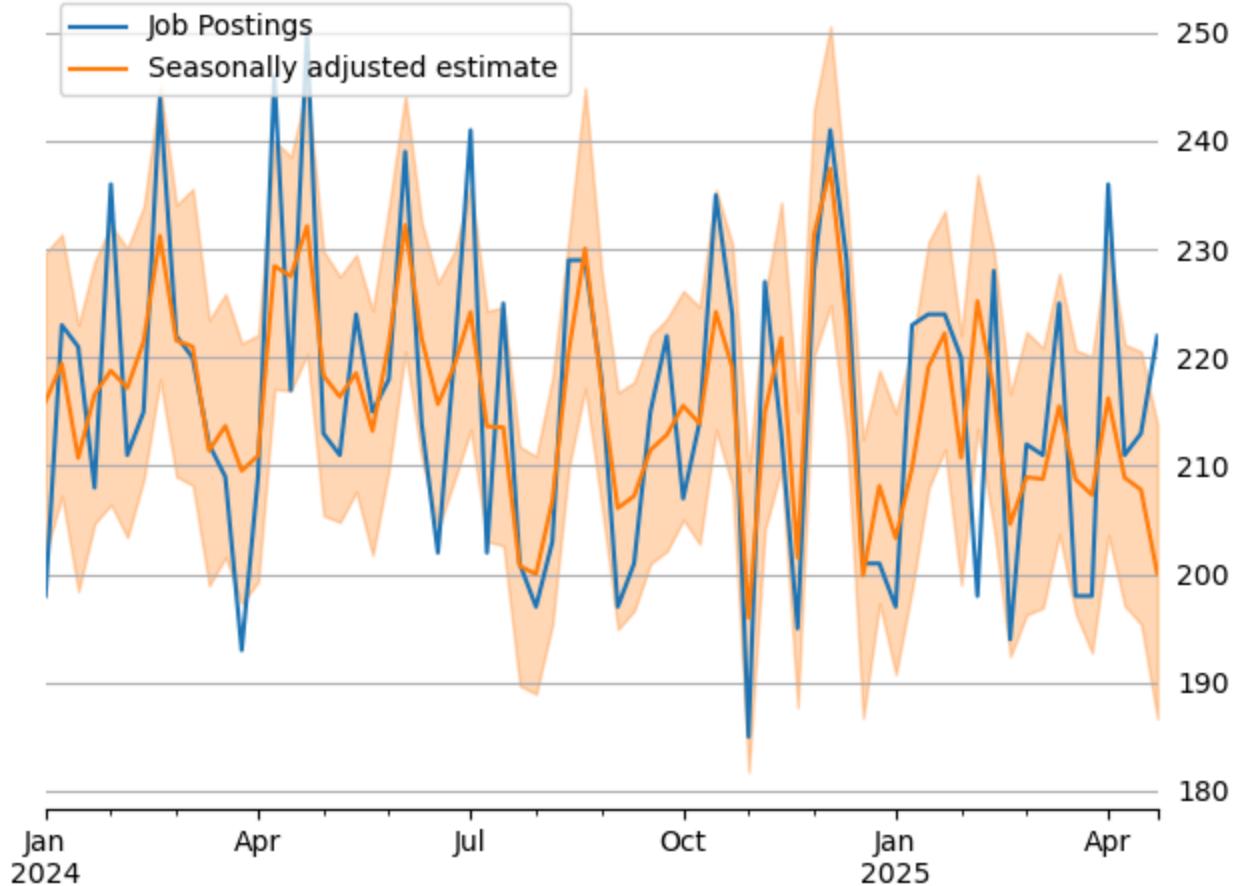
To better illustrate how the seasonally adjusted series eliminates the seasonal patterns, we focus on the five year period from 2014 - 2019:

```
In [52]: fig, ax = plt.subplots()

# Plot the observed data
ax = y.loc['2024-01-07':'2025-04-27'].rename('Job Postings').plot(ax=ax)

# Plot the median and 90/10 credible interval, along with the
q = z.loc['2024-01-07':'2025-04-27'].quantile([0.1, 0.5, 0.9], axis=1).T
q[0.5].rename('Seasonally adjusted estimate').plot(ax=ax)
ax.fill_between(q.index, q[0.1], q[0.9], color='C1', alpha=0.3)

# Finalize the plot
ax.legend(loc='upper left')
ax.set_xlabel(None)
ax.yaxis.set_label_coords(1, 1)
ax.yaxis.set_tick_params(left=False, right=False, labelleft=False, labelright=True)
ax.yaxis.grid()
[ax.spines[s].set_visible(False) for s in ['left', 'top', 'right']]
fig.tight_layout()
```



The analysis indicates that job posting data exhibit strong seasonal fluctuations. However, after removing seasonal effects, the core recruitment demand remains relatively stable, without clear evidence of a long-term trend. This suggests that the labor market is mainly driven by cyclical factors, while the underlying demand level is steady.

### Application 3: computing the "causal impact" of an intervention

One use of Bayesian structural time series (BSTS) models that has grown in popularity is in constructing counterfactual predictions from which to assess the impact of some intervention.

The use of this type of approach in assessing a "causal impact" was described in the Annals of Statistics paper "[Inferring causal impact using Bayesian structural time-series models](#)" (Brodersen et al., 2015), and was popularized via the accompanying `CausallImpact` library in R.

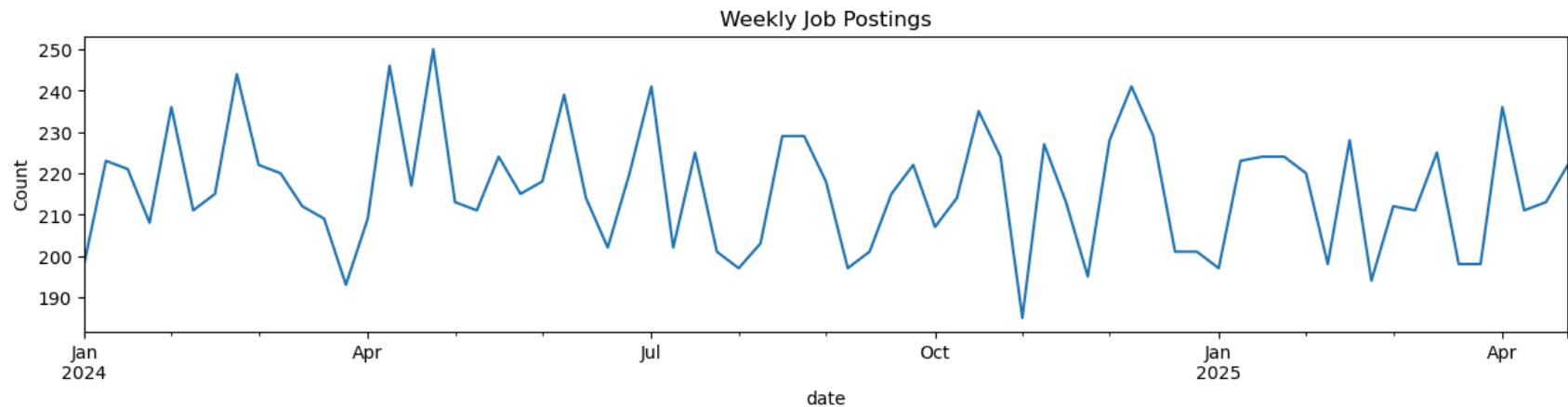
The basic approach to estimating this type of model is the same as above, except that we will only estimate our model over a dataset that ends just prior to the intervention. The forecasts produced by the model can then be compared to the actual data over the subsequent period.

In [53]:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 3))

# Plot only the data between January 7, 2024 and April 27, 2025
y.loc["2024-01-07":"2025-04-27"].plot()
plt.title("Weekly Job Postings")
plt.ylabel("Count")
plt.show()
```



The data shows a clear trend, but it has no obvious seasonal pattern (since it has already been seasonally adjusted). Therefore, we select a local linear trend model.

Since the steps are the same as above, we present them all at once below:

In [54]:

```
import numpy as np
import pandas as pd
```

```
import xarray as xr
import statsmodels.api as sm
from scipy import stats

# Extract training sample (e.g., data up to before 2020-02)
y_train = y.loc[:'2024-10-06']

# Create Local Linear trend model + simulate smoother
mod = sm.tsa.UnobservedComponents(y_train, 'lltrend', use_exact_diffuse=True)
sim = mod.simulation_smoothen(simulate_state=True)

# Forecast steps: full dataset length - training dataset length
n_fcast = len(y) - len(y_train)
ix = y.index.rename('date')
fcast_ix = ix[-n_fcast:] # forecast index range

# Prior distribution
priors = [stats.invgamma(0.001, scale=0.001)] * 3

# Storage containers
niter = 100000
params = xr.DataArray(np.zeros((niter + 1, mod.k_params)),
                      dims=['draw', 'param'],
                      coords={'draw': range(niter + 1),
                              'param': mod.param_names},
                      name='params')

states = xr.DataArray(np.zeros((niter + 1, mod.nobs, mod.k_states)),
                      dims=['draw', 'date', 'state'],
                      coords={'draw': range(niter + 1),
                              'date': mod._index.rename('date'),
                              'state': mod.state_names},
                      name='states')

predict = xr.DataArray(np.zeros((niter + 1, mod.nobs + n_fcast, mod.k_endog)),
                      dims=['draw', 'date', 'variable'],
                      coords={'draw': range(niter + 1),
                              'date': ix,
                              'variable': [mod.endog_names]},
                      name='forecast')

# Initial values: fit with MLE
```



```
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
55000
60000
65000
70000
75000
80000
85000
90000
95000
100000
```

## Constructing the impacts

To construct the impacts, we need to first construct the model predictions. Then the point-wise impacts are equal to the observed data minus the model predictions, and the cumulative impacts are just the cumulative sum of the pointwise impacts.

In [55]:

```
# Parameter settings
start = '2024-01-07'          # Start date of the data
intervention_date = '2024-10-06'  # Intervention point (set by yourself)

nburn = 10000
nthin = 10

# Forecast results
predict_plot = predict.sel(draw=np.s_[nburn + 1::nthin])

# Median and credible interval
predict_median = predict_plot.median(dim='draw').to_pandas().iloc[:, 0]
predict_lower = predict_plot.quantile(0.3, dim='draw').sel(date=np.s_[intervention_date:])[ :, 0]
predict_upper = predict_plot.quantile(0.7, dim='draw').sel(date=np.s_[intervention_date:])[ :, 0]
```

```

# Pointwise prediction differences (observed value - model prediction)
predict_diff = (xr.DataArray(y.to_frame(), dims=['date', 'variable']) - predict_plot)
predict_diff_median = predict_diff.median(dim='draw').to_pandas().iloc[:, 0]
predict_diff_lower = predict_diff.quantile(0.3, dim='draw').sel(date=np.s_[intervention_date:]):[:, 0]
predict_diff_upper = predict_diff.quantile(0.7, dim='draw').sel(date=np.s_[intervention_date:]):[:, 0]

# Cumulative effects
predict_diff_cumul = predict_diff.loc[start: ].cumsum(dim='date')
predict_diff_cumul_median = predict_diff_cumul.median(dim='draw').to_pandas().iloc[:, 0]
predict_diff_cumul_lower = (
    predict_diff_cumul.quantile(0.3, dim='draw').sel(date=np.s_[intervention_date:]):[:, 0])
predict_diff_cumul_upper = (
    predict_diff_cumul.quantile(0.7, dim='draw').sel(date=np.s_[intervention_date:]):[:, 0])

```

Finally, we plot the predictions, point-wise impacts, and cumulative impacts, along with 70/30 credible intervals.

```

In [56]: fig, axes = plt.subplots(3, figsize=(12, 8), dpi=120)

# Observed Values vs. Model Predictions
(y.loc[start: ].rename('Observed data')
 .plot(color='k', ax=axes[0]))

predict_median.loc[start:intervention_date].rename('Model fit').plot(ax=axes[0])
predict_median.loc[intervention_date: ].rename('Prediction').plot(
    ax=axes[0], color='C0', linestyle='--')

axes[0].fill_between(predict_lower.coords['date'].to_index(),
                     predict_lower,
                     predict_upper,
                     color='C0', alpha=0.2)
axes[0].legend(loc='lower right')
axes[0].axvline(intervention_date, color='k', linewidth=1)
axes[0].set_ylabel('Observed Value', loc='top', rotation="horizontal")
axes[0].set_xlabel(None)
axes[0].yaxis.set_label_coords(1, 1)
axes[0].yaxis.set_tick_params(left=False, right=False, labelleft=False, labelright=True)
axes[0].yaxis.grid()
[axes[0].spines[s].set_visible(False) for s in ['left', 'top', 'right']]

# Point-wise impact
predict_diff_median.loc[start: ].rename('Point-wise impact').plot(ax=axes[1])

```

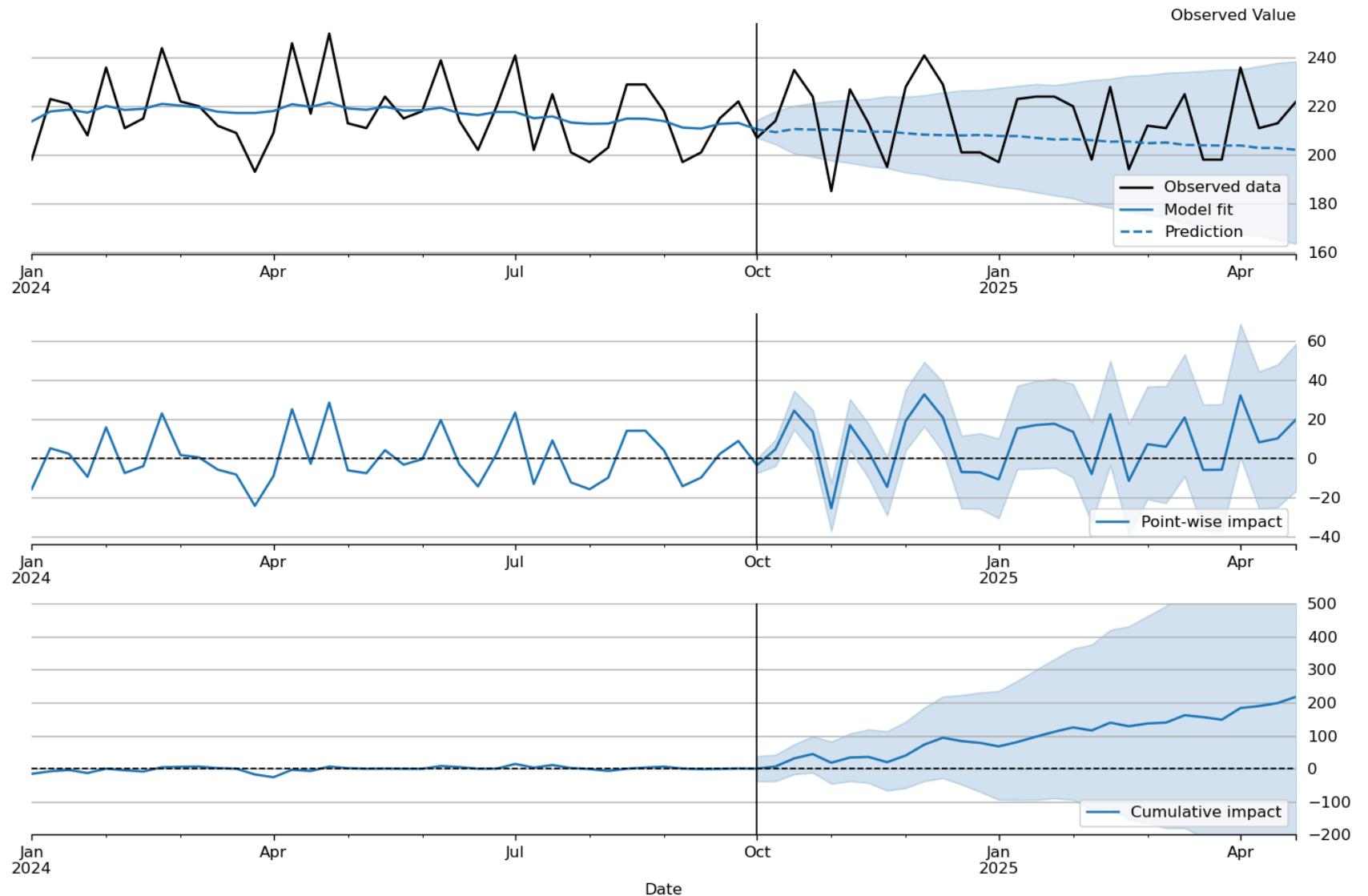
```
axes[1].fill_between(predict_diff_lower.coords['date'].to_index(),
                     predict_diff_lower,
                     predict_diff_upper,
                     color='C0', alpha=0.2)
axes[1].axvline(intervention_date, color='k', linewidth=1)
axes[1].axhline(0, color='k', linewidth=1, linestyle='--')
axes[1].legend(loc='lower right')
axes[1].set_xlabel(None)
axes[1].yaxis.set_tick_params(left=False, right=False, labelleft=False, labelright=True)
axes[1].yaxis.grid()
[axes[1].spines[s].set_visible(False) for s in ['left', 'top', 'right']]

# Cumulative impact
predict_diff_cumul_median.loc[start:].rename('Cumulative impact').plot(ax=axes[2])
axes[2].fill_between(predict_diff_cumul_lower.coords['date'].to_index(),
                     predict_diff_cumul_lower,
                     predict_diff_cumul_upper,
                     color='C0', alpha=0.2)
axes[2].axvline(intervention_date, color='k', linewidth=1)
axes[2].axhline(0, color='k', linewidth=1, linestyle='--')
axes[2].legend(loc='lower right')

axes[2].set_ylim(-200, 500)

axes[2].set_xlabel('Date')
axes[2].yaxis.set_tick_params(left=False, right=False, labelleft=False, labelright=True)
axes[2].yaxis.grid()
[axes[2].spines[s].set_visible(False) for s in ['left', 'top', 'right']]

fig.tight_layout()
```



This analysis applies a Bayesian Structural Time Series (BSTS) model to evaluate causal impact, specifically assessing the effect of an intervention (such as a marketing campaign or policy implementation) on weekly job postings. The results show that the model fits well before the intervention, indicating reliable predictions. After the intervention, job postings were generally higher than predicted, with most weekly impacts being positive. The cumulative effect continued to rise, demonstrating that the intervention significantly increased job postings and generated a sustained positive impact.

